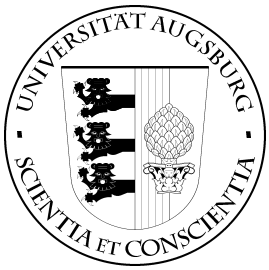
Analysable Instruction Memories for Hard Real-Time Systems

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Fakultät für angewandte Informatik
der Universität Augsburg



Universität
Augsburg
University

eingereicht von
Dipl.-Ing.-Inf. Stefan Metzlaff

Analysable Instruction Memories for Hard Real-Time Systems
Stefan Metzlaff

Erstgutachter:	Prof. Dr. Theo Ungerer
Zweitgutachter:	Prof. Dr.-Ing. Rudi Knorr
Tag der mündlichen Prüfung:	2. Juli 2012

Abstract

In safety-critical embedded real-time systems the timing behaviour is of highest importance, because applications underlie timing constraints that have to be met. Otherwise the system might fail causing harm to humans, the environment, or the system itself. Therefore, such hard real-time systems have to provide timing guarantees. Since the timing behaviour of the system does not only depend on the application itself and is also determined by the particular hardware, there is a need for predictable hardware architectures, as e.g. pipelines or memories. Within this work the focus is on first level instruction memories, which have a large influence on the performance of the system. So a predictable instruction fetch guaranteed by a timing predictable instruction memory hierarchy is crucial for a design of a safety-critical real-time system.

In this work the so called dynamic instruction scratchpad (D–ISP) is proposed and its impact on hard real-time systems is examined. The D–ISP features a function-based dynamic content management that ensures that the currently executed function is always contained in the scratchpad. Hence, it guarantees a predictable and instantaneous instruction fetch, once the active function is loaded. Moreover, the D–ISP eases the timing analysis of the whole system by eliminating the interferences between instruction and data memory access.

For evaluation the D–ISP was verified in a cycle-accurate SystemC model and implemented into an FPGA using the hardware description language VHDL. The D–ISP was integrated into the CarCore processor, which is instruction set compatible to the Infineon TriCore, and it was used as predictable first level instruction memory of the MERASA multicore. An examination of the hardware complexity showed that the D–ISP is more hardware intense than common cache memories, but the costs are in the same order of magnitude. The advantages of the D–ISP implementation compared to a cache are the decoupling of the content and management memory structures and its minimal influence on the timing of the fetch path.

To evaluate the impact of the D–ISP on the WCET estimates of the system a static timing analysis of the D–ISP and the CarCore host processor was performed. It could be shown that in comparison to other common instruction memories like caches and scratchpads with fixed content the D–ISP can provide lower WCET estimates. It can reach an up to 14% lower WCET estimate in comparison to scratchpads with fixed content and reduce the WCET estimate compared to a fully associative LRU cache by at most 29%. Furthermore, different replacement policies for the D–ISP were compared. It is shown that the FIFO replacement policy requires the lowest hardware effort, but cannot reach the WCET estimates that an LRU or a stack-based replacement policy can provide. In general the LRU policy performs best, but it cannot be implemented in hardware. The stack-based replacement policy requires up to 25% more hardware effort than the implementation of the FIFO replacement policy, but reaches similar WCET estimates as LRU.

With the D–ISP a promising alternative to common first level instruction memories for safety-critical real-time systems is proposed. The performed evaluations were able to quantify the impact of the D–ISP on the hardware complexity, the WCET estimates, and the average case performance of the system.

Kurzzusammenfassung

Sicherheitskritische Echtzeitsysteme unterliegen strikten Zeitschranken, die nicht überschritten werden dürfen, da es sonst zu schwerwiegenden Schäden an Menschen, Umwelt oder dem System selbst führen kann. Daher ist es entscheidend, dass die Zeitschranken des Systems garantiert eingehalten werden. Die Ermittlung von sicheren Zeitschranken erfordert die Analyse des Zeitverhaltens des Gesamtsystems, welches aus der Anwendung selbst und der verwendeten Hardware besteht. Eine präzise Bestimmung des zeitlichen Verhaltens von Hardwarekomponenten ist nur möglich, wenn diese so entwickelt wurden, dass sie ein zeitlich vorhersagbares Verhalten zeigen. Daher beschäftigt sich diese Arbeit mit der Betrachtung und Beurteilung von Befehlsspeichern bezüglich ihrer Einsetzbarkeit in Echtzeitsystemen.

Im Rahmen dieser Dissertation wurde ein dynamisches Befehlsscratchpad (D–ISP) als erste Ebene einer Speicherhierarchie entwickelt und dessen Einfluss auf das Zeitverhalten von Echtzeitsystemen untersucht. Das D–ISP verwaltet seinen Speicherinhalt autonom und auf der Basis von Funktionen. Hierzu lädt es die jeweils aktive Funktion aus dem nachgelagerten Speicher und sichert so eine vorhersagbare und unterbrechungsfreie Ausführung von Funktionen zu. Durch die Arbeitsweise des D–ISP garantiert es weiterhin eine Isolation von Befehls- und Datenspeicherzugriffen und ermöglicht so eine Vereinfachung der Analyse des Zeitverhaltens des Echtzeitsystems.

Das D–ISP wurde mittels SystemC taktgenau modelliert und via VHDL in Hardware implementiert. Als Wirtsprozessor wurde der CarCore Prozessor verwendet, welcher einen Teil des Befehlssatzes des Infineon TriCore unterstützt. Weiterhin wurde das D–ISP als echtzeitfähiger Befehlsspeicher im MERASA Mehrkernprozessor eingesetzt. Eine Abschätzung des Hardwareaufwands zeigte auf, dass das D–ISP mehr Ressourcen benötigt als klassische Caches. Im Gegensatz zu Caches entkoppelt das D–ISP jedoch die Strukturen für die Verwaltung und den Inhalt des Speichers und hat nur geringen Einfluss auf das Zeitverhalten des Befehlholepfads.

Zur Bewertung der Ausführungszeit im schlechtesten Fall (WCET) wurde eine statische Analyse des Zeitverhaltens des D–ISPs und anderen gängigen Befehlsspeichern, wie Caches oder Scratchpads mit fester Zuordnung, vorgenommen. Die Ergebnisse zeigen, dass durch die Verwendung des D–ISPs eine Reduzierung der abgeschätzten WCET um bis zu 14% bzw. 29% im Vergleich zu Scratchpads mit fester Zuordnung und Caches möglich ist. Weiterhin wurden drei verschiedene Ersetzungsstrategien für das D–ISP untersucht. Es konnte gezeigt werden, dass die FIFO Ersetzungsstrategie zwar mit geringsten Kosten implementiert werden kann, jedoch die Abschätzungen der WCET von LRU und einer stackbasierten Ersetzungsstrategie in der Regel unterboten werden. Im Allgemeinen ist LRU die beste Ersetzungsstrategie, eine effiziente Implementierung ist aber nicht möglich. Die stackbasierte Ersetzungsstrategie hingegen benötigt bis zu 25% mehr Ressourcen als FIFO, erreicht aber ähnliche WCET Abschätzungen wie LRU.

Es konnte gezeigt werden, dass das in dieser Arbeit vorgeschlagene funktionsbasierte D–ISP Vorteile gegenüber gängigen Befehlsspeichern für Echtzeitsysteme bietet. Die hierzu durchgeführten Evaluierungen beleuchten die Themenkomplexe der Implementierungskosten, des Zeitverhaltens im schlechtesten Fall, sowie der durchschnittlichen Ausführungsleistung.

Danksagung

An dieser Stelle möchte ich mich für die ausgezeichnete Betreuung durch Prof. Theo Ungerer bedanken. Er hat mich nicht nur in jeder Phase der Dissertation von der Suche und Entwicklung des Themas bis zur fertigen Schrift unterstützt, sondern ermöglichte es mir auch meine Ideen im wissenschaftlichen Umfeld bei Projekten und Konferenzen vorzustellen. Weiterhin möchte ich Prof. Rudi Knorr für die Begutachtung meiner Arbeit danken.
Für viele fruchtbare Diskussionen möchte ich mich bei allen Kollegen des Lehrstuhls für Systemnahe Informatik und Kommunikationssysteme bedanken. Besonderer Dank geht dabei an Sascha Uhrig und seinen vielen wertvollen Tipps. Jörg Mische, Irakli Guliashvili und Sebastian Weis möchte ich speziell für die unzähligen Diskussionen über Prozessorarchitektur und deren Implementierung danken.
Meinen Eltern danke ich dafür, dass sie mich jederzeit unterstützt haben. Für Ablenkung und Ausgleich danke ich meiner Familie und allen Freunden. Ganz besonders möchte ich meiner Freundin Melanie Schulz für ihren moralischen Beistand danken.

Stefan Metzlaff, August 2012, Augsburg

Table of Contents

List of Abbreviations

ASIC	Application-Specific Integrated Circuit
BB	Basic Block
BBS–ISP	Basic-Block-Based Static Instruction Scratchpad
CAM	Content Addressable Memory
CFG	Control Flow Graph
CSM	Context Stack Memory
DFA	Data Flow Analysis
D–ISP	Dynamic Instruction Scratchpad
DMA	Direct Memory Access
DRAM	Dynamic Random-Access Memory
ELF	Executable and Linkable Format
FDBP	Function Delimiter Bit Pattern
FIFO	First In First Out
FLE	Function Length Encoding
FMUX	Fetch Multiplexer
FPGA	Field-Programmable Gate Array
FS–ISP	Function-Based Static Instruction Scratchpad
HRT	Hard Real-Time
ILP	Integer Linear Programming
IPET	Implicit Path Enumeration Technique
ISA	Instruction Set Architecture
ISPTAP	Instruction Scratchpad Timing Analysis Program
LRU	Least Recently Used

LTM	Lookup Table Memory
MAT	Memory Access Time
MMAT	Maximum Memory Access Time
MTM	Mapping Table Memory
PLRU	Pseudo-LRU
RAM	Random-Access Memory
SEMC	Separated Off-Chip Memory Connection
SHMC	Shared Off-Chip Memory Connection
SRAM	Static Random-Access Memory
VHDL	Very-High-Speed Integrated Circuits (VHSIC) Hardware Description Language
WCET	Worst-Case Execution Time
WCP	Worst-Case (Execution Time) Critical Path

Chapter 1

Introduction

1.1 Motivation

A wide range of applications in embedded systems like in field of automotive, aerospace, or industrial automation have restrictions on their timing behaviour [Liu, 2000; Bouyssounouse and Sifakis, 2005]. Such systems for which the timing behaviour is of utmost importance are commonly denoted as real-time systems. According to Buttazzo [2005] a real-time system is a computing system that has to react within precise time constraints to environmental events. Thus, the correctness of a system does not only rely on the correctness of the computed result, also the timing of the result is of importance. Basically two different classes of real-time systems can be distinguished [Liu, 2000]: soft and hard real-time systems. For *soft real-time* systems sporadic violations of the timing constraints are tolerable and do not cause a serious system failure. In contrast to this *hard real-time* (HRT) systems do not allow even a single deadline miss, because any violation of the timing equals a fatal system fault that can cause critical damage. Hard real-time systems are necessary in safety-critical environments where timeliness is essential to preclude harm from humans, the environment, or the system itself.

According to [Sangiovanni-Vincentelli and Di Natale, 2007] in automotive systems hard real-time applications are applied to control mechanical systems in chassis or powertrain (e.g. engine, transmission, or emission control) or for active safety functions (like anti-lock breaking system or electronic stability control). Among these application areas powertrain applications are most demanding, for example an engine control has to manage over 100 different engine functions with the help of various sensors and actuators to meet the emission regulations [Cook et al., 2007]. The functions for increasing passengers safety like electronic stability control are much less complex [Cook et al., 2007], but their timing constraints are also of highest importance.

Since such applications for preserving human safety and reducing environmental harms have strict timing constraints, guarantees of the timing for the system have to be given. The estimation of timing guarantees by a timing analysis does not only require a precise knowledge of the application itself, also the particular hardware where it is executed has to be considered [Wilhelm et al., 2008], which consists of the processor, the memory hierarchy, and several peripheral devices (like for communication or interrupts). Hence, there is a need for predictable software design processes, operating systems, and hardware architectures [Lee, 2005]. From architectural point of view this means that timing constraints for e.g. the execution of instructions in a pipeline, the access to the memory, or the communication with a peripheral component have to be provided. The estimation of such timing constraints is only trivial for very simple processors, since in more complex architectures the timing depends on the system state [Wilhelm et al., 2008], like the

previously executed instructions in the pipeline, the content of the cache, or bus contention. Unfortunately, real-time applications are performance demanding such that simple processors without superscalar pipelines or caches do not fit the computational requirements of modern hard real-time applications [Wilhelm et al., 2008].

High performance features in modern processor architectures like out-of-order pipelines, complex cache hierarchies, and branch predictors are designed to improve the average case performance [Hennessy and Patterson, 2006], but in real-time systems the average performance is no valid measure for the architectural design. Since hard real-time systems have to meet their deadlines, the worst-case performance respectively the worst-case execution time (WCET) is the most important system characteristic, beside others that are also of importance like power consumption and package size. Architectural features like some cache implementations or highly speculative pipeline techniques to increase the instruction level parallelism that improve the system's average performance might decrease the worst-case performance of the system. Therefore, a hard real-time system has to be designed to be predictable in the first place. On the architectural level Thiele and Wilhelm [2004] identify the interferences between different processor components as the major source that hinders a predictable design. Such interferences occur when one component influences the behaviour of another. So for example in a unified cache the front- and the back-end of the processor can independently access the cache and alter its content or the accesses can be delayed by each other. Another source for unpredictability is local non-determinism. It is defined as the dependence of a local event, like the timing of the execution of an instruction, upon the global system state respectively the history of former events, like the content of a cache or the state of an out-of-order pipeline. Lundqvist and Stenström [1999] use the term timing anomaly for effects in which a local worst-case behaviour does not even result in a global worst-case behaviour. Anyhow, such systems could be analysed since globally the system is deterministic, but the dependence on a large amount of prior states inflates the analysis complexity, which results in a reduced predictability of the system or in an infeasibly long analysis time.

Two main approaches are distinguished by Thiele and Wilhelm [2004] to reach a predictable system: either to reduce interferences and local non-determinism by design or the establishment of an analysis method that precisely models the behaviour of the system. Plenty of work has been done to provide precise timing analyses for modern processor architectures like for out-of-order pipelines [Li et al., 2004; Rochange and Sainrat, 2009], branch predictors [Colin and Puaut, 2000; Burguiere and Rochange, 2007], or cache hierarchies [Chattopadhyay and Roychoudhury, 2009; Lesage et al., 2009]. On the other hand also hardware architectures that provide a predictable timing and thus mitigate the complexity of timing analysis are proposed, some examples are multicore processors [Hansson et al., 2009; Pitter and Schoeberl, 2010; Ungerer et al., 2010], multithreading processors [Edwards and Lee, 2007; Barre et al., 2008; Lickly et al., 2008; Mische et al., 2010a], on-chip memories [Schoeberl, 2004; Puaut and Pais, 2007; Whitham and Audsley, 2008], memory controllers [Akesson et al., 2007; Reineke et al., 2011], and distributed real-time systems [Kopetz and Grünsteidl, 1993; Kopetz and Bauer, 2003].

Within this work the focus is on first level instruction memories, since they have a strong influence on the performance of the processor and thus a predictable instruction fetch is a foundation for a high worst-case performance. In embedded real-time systems many different instruction memories are used from scratchpads with fixed content to self-managed caches. Since memories with fixed content provide a predictable memory access, but have a decent utilisation, and caches with their fine-grained content management require an expensive analysis, the author has the opinion that memories with a coarse-grained dynamic content management provide the best trade-off between predictability and performance. Therefore, a dynamic function-based instruction scratchpad is proposed that guarantees an instantaneous execution of functions without any instruction fetch delays. Furthermore, it eliminates the interferences between data and instruc-

tion memories sharing the same off-chip memory connection and so increases the predictability of the system according to Thiele and Wilhelm [2004].

1.2 Outline

In this work a first level instruction memory for hard real-time systems is proposed: the so called dynamic instruction scratchpad (D–ISP). The D–ISP features a dynamic content management on the granularity of functions. Therefore, it autonomously loads each function on its activation via call or return. By implementing the whole content management into a hardware controller, the performance of loading and evicting functions is not decelerated by any maintenance routines that are typically used in software-managed scratchpads. The main characteristic of the D–ISP is that it ensures that the active function is always contained in the D–ISP before its execution. Thereby, the D–ISP can guarantee a predictable and instantaneous execution of the function. Also the D–ISP isolates instruction and data memory accesses in a memory hierarchy with shared off-chip memory connection by its design: (1) it stalls the pipeline on loading a function and thus no data memory accesses can interrupt the function load; (2) it assures that every instruction fetch will be served from the D–ISP so that data memory accesses are not interfered by instruction fetches. This two-phased behaviour eases the WCET analysis of the system, since the analysis does not have to take any interferences between the instruction and data memory hierarchy into account. Hence, a separated analysis of data and instruction memory can be performed without losing analysis precision.

The D–ISP was simulated and verified with a cycle-accurate SystemC model and implemented into an FPGA using the hardware description language VHDL. Therefore, the D–ISP was integrated in the real-time capable CarCore processor [Mische et al., 2010a]. Moreover, it was also used as predictable first level instruction memory within the MERASA project [Ungerer et al., 2010]. To evaluate the D–ISP in terms of the impact on the WCET estimates a custom static WCET analysis tool was build, the so called Instruction Scratchpad Timing Analysis Program (ISPTAP). The proposed WCET analysis tool features the CarCore with several instruction memories applied like the D–ISP, multiple static scratchpads, and caches with different replacement policies. The ISPTAP is used to obtain safe WCET estimates of a system with configurable instruction memory and thus it allows to rate the impact of the proposed D–ISP for hard real-time systems.

This thesis contributes to the architectural design of real-time capable instruction memories. Further, it proofs its assumptions and results by hardware cost estimation and quantification of the WCET impact. The major findings of this work are outlined in the following.

I. The proposed D–ISP reaches lower WCET estimates than common instruction memories like static scratchpads or caches. The performed evaluations show that the main source for the lower WCET estimates of the D–ISP is the elimination of interferences between instruction and data memory access. For caches and common scratchpads these interferences can have a large impact on the preciseness of the WCET estimate. An integrated pipeline, data memory, and instruction memory analysis is able provide precise results for these memories, but such analysis tends to be very complex. By guaranteeing that no interference can occur the D–ISP eases the WCET analysis for instruction and data memory while preserving analysis precision. Thus, the D–ISP is a promising alternative to common first level instruction memories used in hard real-time systems.

II. A hardware effort estimation of the D–ISP shows that it more complex than a cache controller, but the D–ISP ensures a predictable execution of the application and provides

also a gain of predictability for data memory access. The organisation of the D–ISP as hardware-managed scratchpad decouples the memory overhead needed to manage the content from the size of the usable memory while keeping the size of the addressable memory blocks. This is not possible for caches which either require larger cache lines or more tag memory, if their size is increased. This characteristic of the D–ISP is of advantage for large scratchpad sizes, because it allows a low amount of memory overhead and internal fragmentation. Moreover, in contrast to a cache the D–ISP does not need a costly hit detection on every instruction fetch, such that it can serve fetches without additional delay.

III. Different replacement policies for the D–ISP are proposed and evaluated. It is shown that FIFO is the replacement policy with the lowest hardware effort, but it has a decent performance especially for small scratchpad sizes. The LRU replacement policy provides in general the lowest WCET estimates for all scratchpad sizes, but it is not implementable in hardware with a reasonable effort. The stack-based replacement policy reaches similar WCET estimates as the LRU replacement policy especially for small scratchpad sizes, while requiring less than 25% additional hardware effort according to the implementation of the FIFO replacement policy.

IV. An average case performance analysis shows that the D–ISP can benefit from the dynamic content management and is able to outperform static scratchpads. However, the D–ISP cannot reach a better average case performance than a cache with its fine-grained dynamic content management.

This thesis is structured as follows. The Chapter 2 provides an overview on related work regarding memories in real-time systems with focus on instruction memories. Also a short introduction on WCET analysis techniques and tools is given in that chapter. The description of the proposed D–ISP from architectural and implementation point of view is provided in Chapter 3. The chapter highlights the features of the D–ISP and their implications on WCET analysis and hardware implementation. The Chapter 4 discusses the analysis of the D–ISP with its different replacement policies. Since in the evaluation the D–ISP is compared to caches and static scratchpads, a short introduction to common cache analysis and WCET-aware code allocation algorithms for static scratchpads is also given. The tool support for the D–ISP is discussed in Chapter 5. It presents a function size instrumentation tool, that is needed by the D–ISP to recognise the sizes of the functions to load. Also the static WCET analysis tool ISPTAP is described, including its capabilities and a validation of its timing model for the CarCore pipeline. Furthermore, this chapter discusses the application requirements of the D–ISP. The evaluation of the D–ISP is presented in Chapter 6. The given evaluation is threefold and covers the hardware effort, the impact on the WCET estimates, and a discussion of average case performance for the D–ISP. Moreover, the complexity of content analyses for the D–ISP and a common cache is compared. The Chapter 7 concludes the work and gives an outlook on future work.

The two appendices provide the complete set of analysis results used by the discussion of the influence of instruction memories on the WCET estimate. Appendix A compares different assignment algorithms for instruction scratchpads with fixed content. The detailed results of the WCET evaluations for the D–ISP are provided in Appendix B.

Chapter 2

Real-Time Capable Memories in Embedded Systems

Embedded systems are typically designed to meet their specifications in terms of performance, manufacturing cost, and energy consumption. Furthermore, for hard real-time systems the timing predictability is a major design criteria. Because the memory subsystem of an embedded system has a strong impact on timing predictability, energy consumption, performance, and cost, it will be in focus of this work.

In this chapter an outline of memories in real-time embedded systems is given. First in Section 2.1 a typical memory hierarchy for embedded systems is discussed to introduce the issues of memory design for embedded real-time systems. In the Section 2.2 the basic terms and function of the different memories in embedded systems is given. The Section 2.3 reviews and discusses different organisations of instruction memories and their usage in real-time systems. For completeness a small introduction to on-chip data memories (Section 2.4) and off-chip memories (Section 2.5) typically used in real-time systems is provided. The basic knowledge of timing analysis is necessary to understand and rate memory architectures for hard real-time systems. Hence, the Section 2.6 introduces common methods to estimate the worst-case execution time (WCET) of applications running on a given system. Finally, Section 2.7 describes the influence of the memory hierarchy design on precision and complexity of the WCET analysis.

2.1 Memory Hierarchies in Embedded Systems

The hierarchical organisation of memories is the classical approach to bridge the gap between large and slow memories to fast but small memory structures [Jacob et al., 2007]. This gap is caused by the different development of the processor and off-chip memory speeds. According to [Marwedel, 2007] the processor speed is increasing by 1.5 to 2.0 times per year, whereas the speed of the off-chip memory increases only by 1.07 times per year. So for embedded systems the impact of slower off-chip memories cannot be ignored, since the clock rates of safety-critical real-time systems are in the magnitude of hundreds of MHz, e.g. for automotive powertrain applications [Sangiovanni-Vincentelli and Di Natale, 2007]. The memory hierarchy of embedded systems is typically less complex as for general purpose systems [Jacob et al., 2007], that use up to three levels of cache, a main memory, and a hard drive.

In the following an exemplary memory hierarchy of an embedded real-time system is to be described. Since hard real-time systems have to be predictable, the memory hierarchy has to be

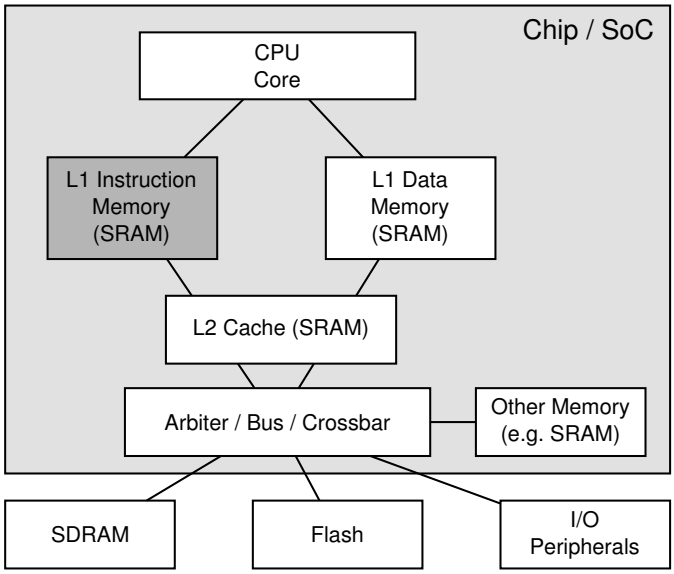


Figure 2.1: Exemplary memory hierarchy of an embedded real-time system

designed accordingly. The hierarchy shown in Figure 2.1 consists of on-chip and off-chip memories. The first level (L1) on-chip memories are separated into an instruction and a data memory, but it is also possible that a unified L1 memory is used. In that case possible interferences in the memory caused by instruction and data accesses impede the timing predictability. Therefore, Wilhelm et al.[2009] suggest the strict separation of data and instruction memory on that level of the memory hierarchy. L1 memories are either organised as caches or as scratchpads. The latter can be managed by software or by special hardware controllers. The concrete organisation of the L1 memory may differ for each microcontroller, even if they are designed for the same application domain. For example the Infineon TriCore and Freescale MPC5554[1], which are both processors for automotive powertrain applications, feature separated L1 instruction and data caches [TriCore, 2004] and a unified L1 cache [e200z6, 2004] respectively.

In addition to the L1 memories embedded systems may also contain other levels of memories as for example an L2 cache, an additional SRAM, or embedded flash memory. The L2 memory can be used to bridge the gap to the slower off-chip memory or to hold often accessed configuration data. The connection to the off-chip memory or peripherals is typically done by buses or crossbars, which can also be hierarchically organised. For example the Freescale MPC5554 uses a crossbar to reach the L2 on-chip memories and peripheral bridges, that are connected to the lower bandwidth peripherals [MPC5554, 2007]. The Infineon TriCore has separated interfaces for the data and instruction memories and uses an additional bus to connect peripherals [TriCore, 2002]. Furthermore, embedded systems can be attached to off-chip memories that provide a much higher capacity than the on-chip resources, like SDRAM or flash memory. The flash memory is used as non-volatile program and configuration storage, whereas SDRAM is typically employed as volatile main memory for data.

It has to be concluded that a typical memory hierarchy of an embedded processor is hard to find, since every manufacturer adds features designed for the specific application domain of

[1]Both processors are among the target processors of the WCET tool challenge 2001 [von Hanxleden et al., 2011].

the processor. Even within the same application domain different memory organisations are common as the two chosen examples, the Infineon TriCore and the Freescale MPC5554, show. For the work presented in this thesis a flat memory hierarchy that consists only of a first level instruction memory and an external SDRAM memory is assumed. This simplification of the memory hierarchy is motivated by (1) the strict focus on first level instruction memories, namely the D–ISP and competing memories, and (2) the observation that no reference memory hierarchy for embedded real-time systems exists. By the view of the author this decision allows it to evaluate the proposed instruction memory without any bias by additional memory components or the memory hierarchy. However, it is supposed that the general findings of this work can be transferred to other memory hierarchy designs.

Beside the simplified memory hierarchy used in this work, more complex multi-level memory hierarchies are also in focus for hard real-time systems, e.g. hierarchies of instruction caches [Hardy and Puaut, 2011], data caches [Lesage et al., 2009], and with unified caches [Chattopadhyay and Roychoudhury, 2009].

2.2 Memory Terminology and Short Introduction

Memory in embedded systems can be classified in memory that is located on the processor chip and memory that is located off-chip. An on-chip memory consists usually of SRAM[2], because SRAM is easy to integrate into the fabrication process of the processor core [Jacob et al., 2007]. Furthermore, it is faster and simpler to access than DRAM[3], because DRAM uses capacitors that have restrictions regarding their charge/discharge times and needs to be refreshed frequently. On the other hand, an SRAM 1-bit cell is more costly than a DRAM 1-bit cell. An SRAM 1-bit cell is typically implemented using six transistors: four transistors to implement two cross-coupled inverters that store the state of the bit and two transistors to access the memory cell[4]. Whereas, a DRAM cell consists of one access transistor and one storage capacitor. The capacitor's charge needs to be frequently refreshed, because leakage currents drain the charge and thus the status of the bit cell will be lost after a given period of time. Since DRAM memories have a lower cost per byte, they are usually employed for large off-chip memories. In real-time systems DRAM memories have an additional downside: the timing of a memory access depends on prior accesses (e.g. by row selection) and the refresh cycle. In Section 2.5 an outline of the relevant work on real-time and DRAM will be given. Beside volatile RAM also flash memory (or disks in general purpose systems) are used as permanent storage, but these memory types are out of the focus of this work. For further reading on memory timing and implementation refer to [Jacob et al., 2007]. In the following the terms *cache* and *scratchpad* are briefly introduced.

A cache is a memory buffer that holds instructions or data to speed up the memory access to the slow non-volatile off-chip memory. Depending on the memory hierarchy multiple levels of cache can be used. The first level cache (L1 cache) is usually located next to the processor core (on-chip), whereas any further level of cache is more likely to be off-chip. With increasing level of cache, the cache capacity and also the access latency increases. The concept of the cache memory gains its benefits from exploiting the temporal and spatial locality of subsequent memory accesses. A cache is usually transparently addressed, i.e. from the application point of view the cache memory is invisible. On a memory access the cache determines whether the requested memory block is contained in the cache or not. On a *cache hit* the requested memory block is returned by the cache. Whereas, on *cache miss* the memory block has to be obtained

[2]SRAM – Static Random Access Memory

[3]DRAM – Dynamic Random Access Memory

[4]There are also implementations with four transistors in total, but they have disadvantages regarding leakage [Jacob et al., 2007].

from the next level of memory, before the request can be served by the cache. A cache basically consists of the *data memory* and the *tag memory*. The data memory is composed of *cache lines* that contain the cached instructions and/or data (depending on the type of the cache). The tag memory keeps a cache tag for each cache line. The cache tag is a part of the address of the memory block stored the dedicated cache line. It is used to determine if a memory block is contained in the cache or not. Caches manage their content autonomously on the granularity of cache lines. The following different cache organisations can be distinguished: *direct mapped*, *fully associative*, and *set associative*. In a direct mapped cache each memory block can be assigned only to one cache line which is typically selected via a modulo operation. The opposite of a direct mapped cache that maps one memory block to exactly one cache line is a fully associative cache, in which every memory block can be placed everywhere in the cache. A set associative cache restricts the possible locations of a memory block to a set of cache lines, e.g. in a 2-way set associative cache a memory block can be located in two cache lines. For associative caches it has to be determined to which cache line a memory block is to be assigned on cache miss. Common (re-)placement strategies are FIFO, LRU, PLRU, and Random. A comparison of the performance of these strategies can be found for example in [Al-Zoubi et al., 2004]. For further information on cache topics like their organisation, optimisation, and implementation the reader is referred to [Hennessy and Patterson, 2006; Jacob et al., 2007].

A scratchpad is a fast on-chip SRAM storage for data and/or instructions. As a cache a scratchpad is used to buffer memory blocks for a faster memory access, but in difference to a cache its content is usually not self-managed. Therefore, it is common to select the scratchpad content by allocation algorithms and copy the selected memory blocks into the scratchpad by software or hardware methods. The copy process is performed either before application start (*static scratchpad*) or during run-time (*dynamic scratchpad*). Typically scratchpads are arranged in the memory map of the processor and thus are directly addressable. The advantage of scratchpads is their energy efficiency, because no energy consuming cache hit detection has to be performed on every memory access. For this reason, scratchpads can be found often in embedded systems. Furthermore, due to the absence of the tag memory a scratchpad provides more usable memory space with the same amount of physical memory. The disadvantage of scratchpads is basically their content management. First, it is not as fast, optimised, and fine-grained compared to the content management of a cache. Thus the impact of a scratchpad on the average case performance is typically lower than for a cache. Second, the scratchpad requires support by the build tool chain to allocate the scratchpad memory and manage the different address spaces of the scratchpad and the main memory. Nevertheless, scratchpads are used in real-time systems, because their content can be defined prior to the execution of the application which eases the analysis of their impact on the timing of the system. A discussion on energy models and the compiler aspects of scratchpads is provided in [Wehmeyer and Marwedel, 2006]. The following section gives an overview of the relevant work on scratchpads and caches with the focus on their usage in real-time systems.

2.3 Real-Time Capable Instruction Memories

2.3.1 Instruction Caches

Instruction caches are common in embedded systems, because they deliver a superior average case performance. Also a lot of effort has been invested to use instruction caches for hard real-time systems, that demand either a good worst-case performance or at least the prediction of the cache behaviour. The challenge of using caches in real-time systems is their dynamic content management. It makes the cache not only content dependent on prior memory accesses, it is

also sensitive to the application's memory layout and task preemption [Mezzetti et al., 2008]. The prediction of the cache content is utmost importance in hard real-time systems, because the content of the cache strongly affects the timing of a whole system, i.e. for the application's execution time it is crucial which memory access results in a cache hit or miss.

The simplest and safe approach for a predictable behaviour is to deactivate the instruction cache. More elaborate methods for real-time systems try to bypass unpredictable access patterns from the cache, lock the cache content, or analyse the cache behaviour when executing real-time applications. In addition to this also special cache structures are proposed that provide predictable cache accesses or mitigate the complexity of cache analysis. In the following an overview of all these aspects for instruction caches in hard real-time systems is provided. A further discussion of the impact of caches on predictability and how to address this issue in real-time systems can be found in [Mezzetti et al., 2008].

Cache Bypassing and Code Positioning

The easiest way to cope with the unpredictable behaviour of caches is to avoid their use, as recapped e.g. in [Shin and Ramanathan, 1994; Jacob, 1999]. But then, every instruction (and data) access is handled by the off-chip memory. Since the gap of processor and memory speed continuously increases [Marwedel, 2007], this is not an option of choice for the high performance demanding real-time systems. So, for example a cache can be activated for predictable access sequences only, such that the performance can be increased while predictability is preserved. In systems with virtual addressing it is possible to decide which page should be cached or not, by maintaining a cached/uncached bit in the entries of the TLB[5] [Chiou et al., 2000]. A more general approach is to split the address space into cached and not cached memory regions. For example in [Plazar et al., 2010] functions with a high impact on the WCET are assigned to a cached memory area. Whereas, functions with a low benefit or functions that have a negative influence on the WCET, when they are using the cache (e.g. due to conflict misses), are allocated to a non-cached memory region.

Mueller et al. intend to increase the predictability of the real execution times and reduce the overestimation of the WCET estimate by forcing cache misses for selected instruction fetches. Therefore, in [Mueller et al., 1994] a *fetch-from-memory bit* is added to the instruction format. If the bit is set, the cache will simulate a miss by fetching the instruction from memory, independently of cache hit or miss. The *fetch-from-memory bit* is set for all instructions for which a cache analysis cannot classify a sure cache hit. Applying this approach results in a higher, but more stable execution time, because unpredictable cache accesses are forced to miss the cache. Also the overestimation of the WCET is reduced, since the system behaves as the pessimistic as proposed by a timing analysis in terms of accesses to the cache.

McFarling identifies instruction collisions in direct mapped caches and proposes an algorithm that solves them by relocation and exclusion from the cache. So the author was able to reduce the overall miss rate of a given program [McFarling, 1989]. The algorithm is not developed for real-time systems, but it can be adapted to solve instruction conflicts and increase the predictability of the system. Then no large run-time variations due to the usage of an instruction cache may occur.

Lokuciejewski et al. propose a compiler optimisation that reduces the cache miss rate by a WCET-aware procedure positioning. Three approaches are presented that optimise the memory layout of the application to reduce the number of conflict misses in the cache [Lokuciejewski et al., 2008a]. Because the optimisation considers the worst-case behaviour of the system, the WCET estimate of the whole application is reduced. Chattopadhyay and Roychoudhury also

[5] TLB – Translation Lookaside Buffer

propose a procedure positioning for real-time systems. But instead of restricting the optimisation to procedures only, the authors propose a holistic approach for code and data positioning in a unified cache [Chattopadhyay and Roychoudhury, 2009]. Other approaches for instruction cache behaviour optimisation by procedure positioning that are not directly addressing real-time systems are for example [Gloy and Smith, 1999] or [Guillon et al., 2004]. But it is supposed that the impact on the WCET is higher for approaches that are especially designed for real-time systems.

In systems with task preemption tasks might mess up each others the cache contents. To address this effect Gebhard and Altmeyer suggest a method that rearranges the code of the tasks such that the intertask interferences in the cache are limited. Therefore, in [Gebhard and Altmeyer, 2007] the start address of each task in a given task set is adjusted. By considering the cache interferences of the different tasks, the effect of preemption in the cache is globally minimised. Another common methods to protect the cache content of different tasks is cache partitioning, which will be discussed subsequently.

Cache Partitioning

The partitioning of the cache can mitigate the effects of cache interferences of different tasks that may preempt each other. In a system with task preemption one task might cause the eviction of the memory blocks from the instruction (or data) cache of the preempted task. This results in an unknown cache state when the preempted task is continued. In the worst case the reactivated task has to reload all memory blocks, because the actual cache state may strictly depend on the cache usage of the preempting tasks. Thus, the WCET estimation of a task requires the knowledge of the conflicting cache set of any preempting task, otherwise no precise timing estimation nor schedulability analysis is possible [Schneider, 2000]. Instead of analysing the cache-related preemption cost on the WCET of a task, as done by Schneider, the partitioning of the cache to isolate the behaviour of the different tasks is easier and promises tighter WCET estimates. For further discussions on scheduling and its effects on the cache behaviour the reader is referred for example to [Staschulat, 2006].

Kirk proposes a hardware-assisted cache partitioning that isolates the cache usage of different tasks. The cache consists of two fractions: a shared pool and the set of cache partitions that are assigned to the tasks [Kirk, 1989]. A small hardware controller decides if the shared pool or the partitions are accessed. Memory accesses of non-critical tasks are always directed to the shared pool. Real-time tasks are allowed to decide whether to use their assigned cache partitions or the shared part of the cache. An additional ID register allows the cache to distinguish which partitions are usable for each task. The assignment of the tasks to their partitions is done statically by an algorithm that takes the characteristics of the tasks and their schedulability into account. The implementation of the proposed cache partitioning is described in [Kirk and Strosnider, 1990].

A software-based cache partitioning is proposed by Mueller. In that approach the cache memory is also separated into a shared part for non real-time tasks and multiple partitions for real-time tasks. The cache partition of a real-time task is defined as a set of cache lines. The number of assigned cache lines may depend on the task's priority and cache needs. To create the cache partitions for the tasks in [Mueller, 1995] the compiler is used to scatter their code (and data) over the address space, such that cache accesses of the tasks only map the designated cache partitions, i.e. the assigned cache lines. Plazar et al. adapt the approach of Mueller and propose an WCET-aware cache partitioning algorithm. The algorithm determines the optimal cache partition sizes minimising the overall WCET of the system by using *integer linear programming* (ILP) [Plazar et al., 2009].

Busquets-Mataix et al. combine cache partitioning and the consideration of cache-related preemption cost during schedulability analysis in a hybrid approach. Thereby, the authors exploit the fact that tasks which use a private cache partition cannot suffer interference from all other preempting tasks [Busquets-Mataix et al., 1997]. When considering the cache interferences of the tasks during the schedulability analysis, it is not necessary that all real-time tasks have to own a private cache partition. Thus, some tasks can share or own a cache partition and the preemption cost that may occur is then taken into account by a *cache response time schedulability analysis* (proposed in [Busquets-Mataix et al., 1996]). This allows a better utilised system, since not all tasks need to have their own cache partition and the cache-related preemption cost is also reduced due to cache partitioning.

Chiou et al. propose a cache partitioning scheme to separate interfering memory accesses in the cache and thus increase predictability. The so called *column caching* allows to dynamically assign cache ways to different tasks by simple enhances of the TLB and the cache replacement logic [Chiou et al., 2000]. This approach is comparable to software-managed scratchpads with the difference that on changes of the memory assignment the content does not need to be explicitly copied into the new memory partition. Paolieri et al. uses a similar method for partitioning a shared L2 cache in a multicore processor [Paolieri et al., 2009b]. Yoon et al. extend this work and propose a WCET analysis that considers the resources of each task [Yoon et al., 2011].

Suhendra and Mitra investigate the combination of cache partitioning and cache locking for a shared cache in a multicore. The authors identify promising combinations of partitioning and locking schemes for real-time systems [Suhendra and Mitra, 2008].

Cache Locking

Cache locking allows to freeze the cache content, such that on a cache miss no eviction occurs and the access is bypassed to the next memory level. So the locking of a cache state increases its predictability, because no unknown or unlikely cache access can disrupt the content of the cache. The implementation of cache locking is rather simple, e.g. by setting a cache control flag that disables the replacement logic of the cache. Before freezing the content of the cache, selected instructions (or data objects) have to be loaded. Two different locking schemes are distinguished in literature: a *static locking*, in which the content of the cache is loaded and locked once, and *dynamic locking*, where the content of the cache can be replaced during run-time.

Puaut and Decotigny propose content selection algorithms for a static instruction cache locking based on the worst-case performance and the system schedulability. The proposed algorithms depend on the worst-case execution time critical path (WCP), that might change during assignment of the instructions to the locked cache content. For reasons of algorithm complexity the authors assume in [Puaut and Decotigny, 2002] a fixed WCP. Therefore, the proposed algorithms cannot deliver optimal cache contents for applications without a single dominating path for all possible cache contents. In [Campoy et al., 2005] this approach is compared with a genetic algorithm. Campoy et al. conclude that both approaches are similar in terms of their results, but the approach of Puaut and Decotigny is significantly faster.

Falk et al. propose a static instruction cache locking for functions. A technique is proposed that assigns the set of functions to the locked cache with the highest impact on the application's WCET estimate [Falk et al., 2007]. In contrast to prior work the proposed algorithm considers changes of the WCP, by recalculating the WCP after the selection of each function. To reach a reasonable run-time for the cache content assignment Falk et al. also propose an algorithm for WCP construction that depends only on precalculated data from a previously performed WCET analysis. Hence, no time-consuming WCET analysis has to be performed on recalculating the WCP.

A dynamic instruction cache locking is proposed by Puaut that allows to load preselected contents to the cache during run-time. Therefore, the tasks are split into regions with an associated cache content [Puaut, 2006]. The cache content of each region is selected depending on the impact of the instructions on the WCET. To respect changes of the WCP during assignment of the instructions, the WCP is regularly updated. On entering a region of a task its specific cache content is loaded and then the cache is locked. Puaut and Pais [2006; 2007] generalise the algorithm for the dynamic instruction allocation to lockable caches and scratchpads. In [Puaut and Pais, 2007] the authors also show the differences between the two memory models and conclude that both approaches have similar effects on the WCET performance.

As for instruction caches, locking is also used to increase predictability of data caches, refer e.g. to [Vera et al., 2003; Vera et al., 2007].

Cache Analysis

The analysis of caches strongly depends on the applied replacement policy. Therefore, the early cache analyses mainly consider direct mapped caches, since their analysis complexity is the lowest. Later when the concepts of cache analysis grew more mature, also set associative caches with different replacement policies could be analysed. Beside correctness and precision the scalability of a cache analysis in terms of higher cache associativities and larger programs is of key interest, since the cache analysis tends to be very complex and time consuming.

An integrated instruction cache and pipeline analysis is proposed by Lim et al. [1994; 1995]. The analysis extends the *timing schema* introduced by Shaw in [Shaw, 1989; Park and Shaw, 1991]. The cache and pipeline are modelled by a so called *worst case timing abstraction*, that selects the paths of the control flow in program constructs (e.g. basic blocks, if-statements, or loops) that can be the worst-case path [Lim et al., 1995]. Therefore, the cache state is tracked by keeping the entry and exit cache content for each program construct. The overall worst-case execution path is build by combining the *worst case timing abstractions* of the program constructs from bottom up by concatenating the associated paths. To join the abstractions of the program constructs timing formulas are applied, which model the timing behaviour of the resulting program construct. Lim et al. propose a direct mapped cache analysis only, but by adjusting the timing formulas the approach could also be used for associative caches. In [Hur et al., 1995] the approach is used for seven small benchmarks running on a RISC processor with direct mapped instruction cache showing that the analysis safely upper-bounds the WCET. Hur et al. also state that the cache analysis can be easily parametrised to configure size, line size, and the associativity of the cache.

Mueller and Whalley established an analysis of a direct mapped cache using iterative data flow analysis [Mueller and Whalley, 1994; Mueller, 1994]. This so called *static cache simulation* categorises the cache access for every instruction for all unique paths in the application. The accesses are assigned to one of the following classes: *always-hit*, *always-miss*, *first-miss*, and *first-hit*[6]. Then the obtained cache categorisations can be used to calculate the WCET. This is done in [Arnold et al., 1994]. The authors show that a system with cache is predictable and can provide significantly lower WCET estimates than a system without cache. Mueller describes in [Mueller, 2000] the abstraction of all reachable *concrete cache states* during data flow analysis to limit the needed memory and reduce the analysis time. The following cache states are proposed to allow the classification of the cache accesses into the aforementioned categories: *abstract cache states* (contain all memory blocks[7] that may be in the cache), *dominator cache states* (contain

[6]The original publication does not contain the *first-hit* class, but the following work [Arnold et al., 1994; Healy et al., 1999] introduced it.

[7]Memory block – a continuous sequence of memory that is mapped to a cache line. Also denoted as *program line* [Mueller, 2000] or *l-block* (line-block) [Li et al., 1995]

all memory blocks that have to be in the cache), and *linear cache states* (are used to determine, if a memory block is in the cache, because of a loop or a sequential control flow). Furthermore, the work of Mueller adds the support for the analysis of set associative LRU caches [Mueller, 2000]. It is stated that the cache analysis scales well, but only rather small and regular benchmarks are considered in [Arnold et al., 1994; Mueller, 2000]. Healy et al. enhances the analysis to also consider the pipelining effects that for example occur when multi-cycle instructions like floating-point operations are supported [Healy et al., 1999]. Refer to Section 2.6 for a closer look on pipeline analysis.

Li et al. propose a cache analysis for direct mapped caches using *cache conflict graphs* [Li et al., 1995]. A *cache conflict graph* models for each line of the direct mapped cache the dependencies of all blocks that are mapped to this line. Therefore, the graph contains all control-flow-triggered transitions that cause the loading, the keeping, and the eviction of the conflicting cache blocks. The work is extended by [Li et al., 1996] to a 2-way set associative LRU cache by enhancing the *cache conflict graphs* to *cache state transition graphs* that model all possible states for each cache line during program execution. According to Li et al. the analysis can also be used for larger associativities and other replacement policies. But it is supposed that the *cache transition graphs* will gain in complexity with higher associativities, causing a high memory usage and long analysis run-times.

In [Cassé and Sainrat, 2006] the cache analysis approaches of [Li et al., 1995] and [Healy et al., 1999] are compared for a direct mapped instruction cache with the result that the WCET estimation using the *cache conflict graphs* of Li et al. is more accurate. Whereas the time needed by the analysis of Healy et al. is noticeable smaller, due to the reduced complexity of the formulated linear program.

To overcome the drawbacks of high complexity and long analysis times in [Alt et al., 1996] a cache analysis is proposed that uses abstract interpretation [Nielson et al., 1999]. The work is extended by Ferdinand et al. to use *must* and *may* sets that determine sure hits and misses respectively [Ferdinand et al., 1997; Ferdinand and Wilhelm, 1999]. The cache analysis of Ferdinand and Wilhelm [1999] is state of the art for analysing an LRU cache. For a detailed description of the analysis of Ferdinand and Wilhelm refer to Section 4.2. Furthermore, Theiling et al. propose a separated cache and pipeline analysis to increase analysis speed. In [Theiling et al., 2000] also the persistence analysis of [Ferdinand and Wilhelm, 1998] is added to the instruction cache analysis, that allows it to categorise memory blocks, which are loaded only once and then never get evicted from the cache (similar to the *first-miss* categorisation). An improvement of the *first-miss* detection in terms of precision and analysis time is proposed in [Ballabriga and Cassé, 2008a]. Huynh et al. identify a problem with the safety of the persistence analysis of [Ferdinand and Wilhelm, 1998] and provide a solution [Huynh et al., 2011].

Reineke et al. give an overview of several common cache replacement strategies and their effect on predictability [Reineke et al., 2007]. The authors conclude that the LRU replacement policy is the most predictable replacement policy. Unfortunately, the caches in real processors are not always using LRU, because of higher complexity than other replacement policies as FIFO or PLRU[8]. In [Grund and Reineke, 2010a] three recently proposed methods of FIFO cache analysis using relative competitiveness [Reineke and Grund, 2008], early-miss exploitation [Grund and Reineke, 2009], and phase detection [Grund and Reineke, 2010a] are compared. It is shown that by this analysis techniques the precision of the cache analysis can be significantly increased and a precise FIFO cache analysis is made possible. Refer to Section 4.2 for a deeper insight into the challenges of the analysis of a FIFO replacement policy.

[8]PLRU — Pseudo-LRU, a tree-based approximation of the LRU replacement policy, with reduced hardware complexity, see [Al-Zoubi et al., 2004]

As for FIFO, the analysis of the PLRU replacement policy is not as easy as for LRU [Berg, 2006]. But also different analyses approaches are presented in [Reineke and Grund, 2008] and [Grund and Reineke, 2010b]. Reineke and Grund [2008] propose an analysis that delivers upper bounds for the number of misses and lower bounds for the number of hits based on the relative competitiveness to the sound LRU cache analysis. Whereas, another approach of Grund and Reineke exploits the tree organisation of the PRLU replacement policy to provide a preciser cache analysis. In [Grund and Reineke, 2010b] it is also shown that in general an analysis using the proposed abstraction of the PLRU cache states can guarantee higher hit rates and thus enables tighter WCET estimates.

Metzner propose a WCET analysis including instruction cache analysis by using a model checking approach. The usage of timed automata with concrete cache semantics can mitigate the pessimism that arises when joining paths in a cache analysis using abstract interpretation [Metzner, 2004]. But it is also shown that the model checking approach increases the analysis time. Another model checking based approach for WCET analysis of an instruction cache is proposed by Lv et al. The authors model the cache by transferring the program's control flow into a timed automaton that simulates the programs behaviour including system characteristics [Lv et al., 2010b]. To cope with the state space explosion that inherits the model checking approach the authors cut branches that cannot affect the worst-case execution behaviour of the program. Another model checking based WCET analysis with the support of caches is described in [Dalsgaard et al., 2010].

Chattopadhyay and Roychoudhury use model checking to refine the results obtained by a cache analysis based on abstract interpretation. As already stated in [Metzner, 2004], cache analysis using abstract interpretation loses precision on joining control flows. The work presented in [Chattopadhyay and Roychoudhury, 2011] directly addresses this source of impreciseness by using a model checking cache analysis to remove falsely determined cache conflicts of the classic cache analysis using abstract interpretation that is performed in a previous step. Therefore, cache accesses that could not be classified by abstract interpretation (which are marked as NC) are passed to a preciser model checking analysis. It detects, if the conflict actually occurs, or if it is caused by infeasible paths. So it is possible to identify spurious cache conflicts and increase precision of the analysis. Since the model checking is only invoked for cache conflicts and the proposed method works within a given time budget, the high analysis time of model checking is upper bounded. Furthermore, the use of model checking only refines the results of the cache analysis using abstract interpretation. Hence, the analysis provide always safe WCET estimates, even if the model checker refinement is terminated because it violates its time budget. Providing a larger time budget to the analysis, allows the model checking to further improve the quality of the estimates.

Hardy et al. address the long analysis times of conventional cache analysis (e.g. by abstract interpretation [Ferdinand and Wilhelm, 1999] or static cache simulation [Mueller et al., 1994]) for large programs. The authors propose in [Hardy et al., 2011] a fast and scalable instruction cache analysis, that is in fact slightly less precise, but requires significantly lower analysis time and memory. Instead of taking the whole program for cache analysis into account the basic analysis considers cache effects only within basic blocks and loops. Two extensions of the basic analysis take the cache effects of memory blocks into account that are shared between multiple basic blocks and that are accessed in different call contexts of the same function. The evaluation in [Hardy et al., 2011] shows that a much simpler cache analysis can reach a similar precision as conventional approaches using abstract interpretation.

The idea of speeding up analysis by splitting the cache analysis is similar to the approach proposed in [Patil et al., 2004], that analyses the cache content on a per module/function basis using static cache simulation. In a first step the cache access categorisations are created on a

module level. Then, the second step combines the results from the module-level analysis in a bottom-up manner. Since the approach of Hardy et al. is simpler (especially when the basic analysis is applied without any extensions), it is supposed that it outperforms the approach of Patil et al. in terms of analysis time[9]. Further approaches to speed up the cache analysis by partitioning the analysis are given in [Rakib et al., 2004; Ballabriga et al., 2008].

Analysis of Multi-Level Caches

Beside the analysis of single level caches, different approaches for multi-level cache hierarchies are proposed. The approach of Mueller and Whalley was extended to multi-level cache hierarchies in [Mueller, 1997]. Hardy and Puaut propose an analysis of multi-level instruction caches. Their analysis of higher level caches is performed by using the *hit/miss classification* and the *cache access classifications* of the memory accesses from the lower cache level [Hardy and Puaut, 2008]. In [Hardy and Puaut, 2011] the method is extended to add support for inclusive and exclusive cache hierarchies and other replacement policies than LRU. Lesage et al. add in [Lesage et al., 2009] data cache support to the approach of Hardy and Puaut. The analysis of unified caches in a cache hierarchy is proposed by Chattopadhyay and Roychoudhury. The authors also show in [Chattopadhyay and Roychoudhury, 2009] the analysis of different cache hierarchies consisting of instruction, data, and unified caches.

Cache Analysis in Multicore Processors

The cache analysis of multicore processors is more complicated than for single-core processors, because when using shared caches (e.g. L2 caches) different tasks running concurrently on the cores may affect the state of the shared cache (inter-core interferences). If no cache locking or partitioning is used (as e.g. proposed by [Suhendra and Mitra, 2008], [Paolieri et al., 2009b], or [Yoon et al., 2011]), such interferences in the shared cache have to be modelled.

Yan and Zhang present an analysis of shared instruction caches for multicore processors. By identifying the interferences of different cores in the instruction cache in the worst case a static analysis is made possible [Yan and Zhang, 2008]. To provide tighter WCET estimations for a multicore with a shared L2 instruction cache Hardy et al. use compiler support to explicitly bypass conflicting instructions that are known to be accessed only once in the shared L2 cache [Hardy et al., 2009]. The cache analysis itself is based on the work of Hardy and Puaut [2008] for multi-level caches and is extended to model the inter-core interferences of a shared L2 cache.

Li et al. also propose a shared L2 instruction cache analysis for multicore processors based on identifying possible conflicting cache accesses from different cores. The analysis extends prior methods by identifying inter-core interferences at task-level, i.e. it takes the lifetimes of the tasks into account and rules out cache interferences of tasks which are not executed concurrently on different cores [Li et al., 2009].

Chattopadhyay et al. combine the analysis of shared caches with the analysis of the shared bus in a multicore. The analysis of the shared L2 cache determines the inter-core interferences of co-running tasks. For bus analysis a TDMA[10] arbitration scheme is considered. Since cache and bus analysis depend on each other [Chattopadhyay et al., 2010], both analysis are integrated in an iterative analysis process.

[9] The approach of Patil et al. is able to reduce the analysis time for larger programs only, because it adds some additional complexity to the module-level analysis, that doesn't pays off for small applications. Whereas, the approach of Hardy et al. is faster for all examined benchmarks, but it trades the increase of analysis speed with a loss of precision.

[10] TDMA – Time Division Multiple Access

Gustavsson et al. propose a WCET analysis for a multicore including private L1 (instruction and data) and shared L2 caches using model checking. Therefore, for each component and also the program a timed automaton is created. Due to the size and complexity of this model the analysis is very costly in terms of analysis time and memory usage [Gustavsson et al., 2010]. The authors propose ideas to mitigate this problem e.g. by optimising the search for the WCET estimate, eliminating redundant paths, or creating a coarser grained, but still precise, model of the system. Lv et al. also propose a timing analysis tool that models the shared resources of a multicore like bus and L2 cache using timed automata [Lv et al., 2011]. In contrast to Gustavsson et al. the private caches are analysed using abstract interpretation [Lv et al., 2010a].

Randomised Cache

Instead of categorising cache accesses or increasing predictability of a cache Quiñones et al. propose a randomised cache replacement policy for real-time systems. The work is motivated by the presence of cache access patterns for common caches that result in a pathological worst-case cache behaviour. By using a truly randomised cache the probability of the occurrence of such pathological worst-case behaviour is made independent of any concrete cache access pattern [Quiñones et al., 2009]. Therefore, from probabilistic timing analysis[11] point of view such cache is more desirable for real-time systems than a cache with deterministic replacement policy. However, applying a randomised cache in hard real-time systems is not useful, since the worst case is neither alleviated nor made impossible. A precise description of the worst possible cache behaviour (e.g. by a precise cache analysis) or a cache that mitigates the worst case by design is much more advisable for hard real-time systems.

Schoeberl's Method Cache

Schoeberl proposes in [Schoeberl, 2004] a predictable function-based *method cache*, that uses complete methods as replacement granularity. Whenever a method is called or returned the according method is completely loaded into the cache. That alleviates the analysis of the cache content and allows a precise cache hit/miss prediction. The method cache features a FIFO replacement policy in which methods are loaded or evicted on the whole. The methods itself are stored within one or more memory blocks. The implementation of the method cache employs a cache-like structure that binds each memory block to a cache tag. So the usage of smaller memory blocks to improve the memory density, leads to a high number of cache tags that are causing either a slow or hardware-intensive hit detection. In contrast to the method cache the proposed D–ISP uses lookup tables for content management and thus the scratchpad content is decoupled from the lookup structure. Hence, the complexity of the D–ISP's hit detection depends on the number of lookup table entries that are checked in parallel and not on the size of the memory. Preußer et al. address the complexity problem of a fine-grained method cache implementation and show an implementation with a stack-based replacement policy [Preußer et al., 2007].

In [Huber and Schoeberl, 2009] Schoeberl's method cache with FIFO replacement policy is modelled for WCET analysis[12]. Three approaches for WCET analysis are compared by Huber and Schoeberl: first assuming that every cache access is a miss, second an IPET-based[13] model of the method cache, and third the modelling of the method cache using timed automata.

[11] Refer to the work of Bernat et al. [2002; 2005]

[12] In [Schoeberl and Pedersen, 2006] a simplified version of the method cache was analysed. An attempt to use abstract interpretation to model the method cache [Kirner and Schoeberl, 2007] has proven to be too optimistic.

[13] IPET – Implicit Path Enumeration Technique

The first approach charges for every call and return the cost of loading the activated method into the cache. This is the most pessimistic but safe approach to estimate the WCET for the method cache.

The IPET-based method cache analysis does not use data flow analysis (DFA) to classify every cache access, instead in [Schoeberl et al., 2010] the maximum number of possible cache misses when executing a method is determined. This is possible due to the fact, that on access of a method a known number of cache blocks are accessed at least once. The number of cache blocks that might be accessed when executing a method is determined by aggregating the number of blocks needed by all reachable methods including the one under observation. If all needed blocks fit the method cache, it can be safely assumed that they will be loaded at most once. This results in a more precise estimation than assuming always a miss. Because of the fact that the prior state of the method cache is not accurately considered for the analysis of a certain method, the analysis will introduce some pessimism to the results, e.g. if the blocks that are assumed to miss only once, are already in the method cache from any prior method call. Furthermore, by assuming that every child method might be invoked when analysing a certain method, disjoint execution paths that call different methods are not modelled correctly by this approach. This will lead to a higher overestimation for applications in which different methods are invoked depending on the control flow of their parent method.

The third approach is a simulation of the method cache by using model checking to address the impreciseness of the IPET-based analysis. The model checking approach is based on a simulation of a network of timed automata. Indeed, this approach is much more expensive in terms of run-time than IPET, but it is easier to build precise timing models of architectural features[14]. The method cache is modelled as a set of global variables, that represent the cache blocks, as described in [Huber and Schoeberl, 2009]. Then on invocation of call and return the model checks the state of the method cache. If the activated method is not in the cache, a load penalty is charged for this invocation. The behaviour of the method cache is simulated by implementing its state and its content management into the timing model. To determine the WCET of a program it has to be transformed to an automaton, see for example [Metzner, 2004]. Then the network of timed automata, which also contains the model of the pipeline and the method cache, can be simulated. A detailed description of this approach is given by [Schoeberl et al., 2010].

The three described approaches to obtain a WCET for the method cache are compared in [Huber and Schoeberl, 2009] and also in [Schoeberl et al., 2010]. The resulting WCET estimates of the different approaches are very similar: the preciser model checking approach reaches only a 2-7% lower WCET compared to the IPET-based approach, as shown in [Schoeberl et al., 2010]. This small difference between the approaches can be caused by the small cache miss cost (in both papers 2 cycles for memory read are specified). A larger memory access cost might unveil the different levels of preciseness of the compared analysis techniques.

Schoeberl et al. [2010] state that a DFA of the method cache is too costly, even for small benchmarks. This assumption cannot be confirmed by the analysis of the D–ISP performed in Chapter 6. But nevertheless, the findings of Section 6.4 show that such analysis is not scalable for larger applications.

2.3.2 Static Scratchpads

Scratchpad memories are mainly used in embedded systems to reduce energy consumption or lower the WCET of hard real-time applications. Both goals can be reached, because scratchpads

[14] See for example [Gustavsson et al., 2010] in which a multicore with L1 caches that supports spinlock-based synchronisation is analysed.

(usually SRAMs) have a low and constant memory latency and do not need an energy consuming tag memory like caches. The content of scratchpads can be assigned either statically or it is dynamically managed during run-time by software or hardware. From memory content point of view static and dynamically managed scratchpads behave like statically and dynamically locked caches [Falk et al., 2007; Puaut and Pais, 2007] respectively with essential differences in their implementation and energy consumption. In this section the static scratchpads are discussed in detail. Related work on software- and hardware-managed scratchpads is provided by the two following sections.

Usually the content of a static scratchpad is defined by the application designer before compile-time. This simplifies the analysis of the system's timing behaviour, because already before application execution it is clearly known what is located in which memory and furthermore this assignment does not change. Hence, the usage of scratchpads with a static assignment of instructions allows exact timing estimations for the memory accesses and eases the computation of the WCET. The drawback of static assignment is that the scratchpad memory content cannot change during run-time. Thus, in contrast to caches the memory utilisation of statically assigned scratchpads is poor. However, a good average performance, a reduced energy consumption, or lower WCET estimates are reachable with appropriate scratchpad assignment algorithms.

Banakar et al. propose the usage of scratchpads for embedded systems to lower their energy consumption. A knapsack algorithm is employed to assign the code (and data) to the scratchpad that is most frequently accessed [Banakar et al., 2002]. The authors show that the scratchpad can outperform a cache in terms of energy consumption and area/performance ratio. Steinke et al. extends the work by applying an energy-aware scratchpad content selection for code (and data) [Steinke et al., 2002a]. The ILP-based algorithm selects whole functions, basic blocks, and variables with the highest potential regarding energy saving and assigns them to the scratchpad. It also considers the additional energy cost for adding jumps to reach the and return from the relocated code in the scratchpad. This algorithm was used by Wehmeyer and Marwedel to quantify the impact of scratchpads on the WCET. In [Wehmeyer and Marwedel, 2004] it is shown that a static scratchpad can improve the WCET of a system, even if the memory assignment is performed by an algorithm optimising the energy consumption. In [Wehmeyer and Marwedel, 2005] the impact of a static scratchpad and a cache on the WCET estimate is compared. The authors use the scratchpad allocation algorithm of [Steinke et al., 2002a] and a direct mapped unified cache for their study. For WCET analysis the static timing analysis tool aiT [Heckmann and Ferdinand, 2006] was used. Wehmeyer and Marwedel could show that a scratchpad can deliver lower WCET estimates than a cache[15]. Unfortunately, since the used assignment algorithm is based on optimising the energy consumption, not the minimal reachable WCET estimate for a static scratchpad can be determined by the applied algorithm. In [Wehmeyer and Marwedel, 2006] a more elaborate comparison of scratchpads and caches is provided.

Altering the solution of the knapsack problem to use the benefit on the worst-case timing instead of the possible energy reduction as decision criteria for assigning code to a scratchpad, allows a more directed reduction of the WCET estimate by using a static scratchpad. Unfortunately, such knapsack-based assignment algorithm cannot deliver the optimal scratchpad assignment, since it is not able to take changes of the WCP caused by the scratchpad assignment into account. In [Sepp, 2009] a WCET-aware knapsack-based scratchpad assignment algorithm is described. Furthermore, Sepp shows how the granularity of the assigned code affects the WCET estimate by comparing the assignments found with a granularity of functions and basic

[15] In [Wehmeyer and Marwedel, 2005; Wehmeyer and Marwedel, 2006] it is stated that the cache analysis that was performed inherits some pessimism due to only a subset of the possible analysis techniques could be applied and the use of an unified cache. Nevertheless, as also shown in Appendix B, static scratchpads are able to outperform caches in terms of WCET estimates.

blocks[16]. A more precise discussion of a WCET-aware knapsack-based scratchpad assignment will be discussed in Section 4.1.

Angiolini et al. propose a static scratchpad assignment algorithm that selects the code to reduce energy consumption or to maximise execution performance [Angiolini et al., 2004]. The used algorithm is based on an algorithm introduced by [Angiolini et al., 2003], but it employs different cost and benefit functions. To optimise the performance an application trace is generated and utilised by the algorithm. If the worst-case timing instead of the average case performance would be taken into account, the approach of Angiolini et al. could also be used to optimise the WCET. But this requires the profiling of the worst-case application behaviour instead of tracing the execution of the application.

Whitham and Audsley propose a trace scratchpad that holds program traces like a trace cache, but is not dynamically updated during run-time [Whitham and Audsley, 2008]. The trace scratchpad bypasses the traditional fetch path and the decode stage, which is advantageous in terms of power consumption and performance. Therefore, it has to contain decoded microoperations, that directly control the functional units of the processor. Whitham and Audsley propose a WCET-aware algorithm to find the trace with the highest impact on the WCP and assign it to the trace scratchpad. The trace scratchpad is activated on receiving a *start-of-trace* microoperation. Then instead of fetching and decoding, the instruction stream is executed from the trace scratchpad. If the trace is left, which is signalled by a *end-of-trace* microoperation, the conventional fetch and decode path is reactivated. Whitham and Audsley were able to show a reduction of the WCET estimates by assigning selected traces to the proposed scratchpad. Their results further illustrate that a combination of a conventional (static) instruction scratchpad and the trace scratchpad is beneficial, because it allows to reach lower WCET estimates than using either one or the other scratchpad separately. In [Whitham and Audsley, 2008] only the static assignment of the trace scratchpad is considered, but it is possible to replace the content dynamically either by software or hardware.

Falk and Kleinsorge are using the ILP formulation of [Suhendra et al., 2005] as foundation for their WCET-aware scratchpad assignment for basic blocks [Falk and Kleinsorge, 2009]. Suhendra et al. propose a WCET-sensitive scratchpad allocation for data. Therefore, an ILP formulation is presented that finds an allocation while taking changes on the WCP caused by assigning data object to the faster scratchpad into account. Consequently the proposed solution delivers the optimal data object assignment to the scratchpad while minimizing the WCET of the application. This is in contrast to a formulation of the assignment as knapsack problem that will not necessarily find the optimal solution, since it cannot take any WCET-critical path changes into account (refer to Section 4.1). As the work of Suhendra et al., the assignment algorithm of Falk and Kleinsorge is aware of WCP changes and thus delivers an optimal selection of the basic blocks to reach a globally minimised WCET estimate of the system. Therefore, the approach takes additional penalties of connective jumps and increased basic block sizes caused by additional jump instructions into account [Falk and Kleinsorge, 2009; Falk and Lokuciejewski, 2010]. This WCET-aware scratchpad allocation of basic blocks will be discussed with more detail in Section 4.1 and was selected as optimal case for static scratchpads with basic block assignment (BBS–ISP) in the evaluation presented in Section 6.2. An adaptation of this approach that allows only functions to be assigned will be also described in Section 4.1.

[16]This study is extended by the comparison of different static scratchpad assignment algorithms presented in the Appendix A.

2.3.3 Software-Managed Scratchpads

In contrast to static scratchpads software-managed scratchpads allow the changing of the content during run-time. This enables a better utilisation of the scratchpad memory space. But the downsize is that the memory content has to be explicitly copied into the scratchpad during run-time. This has a non-negligible timing overhead, especially when the content is loaded by software routines. Furthermore, a timing analysis of a system with a software-managed scratchpad has to take the content changes and the copy process into account. But due to the limited number of copy point which are also known a priori, an analysis of a software-managed scratchpad is less complex than the analysis of a cache.

Steinke et al. extend the energy-aware assignment algorithm of [Steinke et al., 2002a] for static scratchpads to dynamically change the scratchpad content. Since an application can contain multiple phases in which different code is executed, multiple copy points at which the code in the scratchpad may be reassigned are defined in the control flow [Steinke et al., 2002b]. An ILP-based algorithm selects for each copy point the code parts with the highest impact on the energy consumption and assigns them to be loaded into the scratchpad by a copy function. Steinke et al. show for a set of small benchmarks that this approach is able to reach a better performance and a lower energy consumption than a cache. Verma et al. propose a dynamic allocation algorithm for instructions and data objects that is based on global register allocation algorithm typically used in compilers [Verma et al., 2004]. The allocation algorithm is designed to reduce the energy consumption, but in [Wehmeyer and Marwedel, 2006] it is shown that this algorithm is also able to reduce the WCET estimates. Furthermore, the algorithm of Verma et al. [2004] is able to provide lower WCET estimates than the energy-aware static scratchpad allocation proposed in [Steinke et al., 2002a]. In [Verma and Marwedel, 2006] the algorithm is enhanced to allow inter-procedural optimisations. Wehmeyer and Marwedel [2004; 2005; 2006] show that (static and dynamic) scratchpad allocation algorithms which are designed to reduce the energy consumption of the memory system can also improve the worst-case timing of a system. Nevertheless, a WCET-aware assignment algorithm will surely provide a better WCET performance, than an algorithm that does not consider the worst-case timing.

In [Ravindran et al., 2005] a compiler-managed dynamic instruction scratchpad is proposed. The scratchpad allocation algorithm uses a greedy heuristic that takes the energy consumption into account to select the most beneficial code parts to be mapped to the scratchpad. The algorithm assigns the code on the granularity of basic block sequences, also denoted as traces. The optimal placement of the selected traces within the scratchpad is determined by the use of *temporal ordering* introduced by Gloy and Smith [1999] – a metric similar to the one proposed by Janapsatya et al. [2006a] that will be discussed in the next section. Ravindran et al. insert special copy instructions into the code to manage the content of the scratchpad during run-time. To load code into the scratchpad the copy instruction requires the source and target address and also the size of the code that is to be loaded. The authors suggest to implement the copy instructions either by software interrupts or by using a DMA unit. Unfortunately, neither the run-time nor the hardware overhead of both approaches is quantified in [Ravindran et al., 2005]. A comparison of the proposed dynamic algorithm, a static variant of it, and a cache shows that the dynamic scratchpad assignment algorithm reaches the highest energy reduction.

Egger et al. propose an instruction scratchpad, that combines static and software-managed approaches [Egger et al., 2006]. At compile time the functions are divided into three classes and located to different memory locations. Frequently used functions are placed in the scratchpad. Whereas, rarely called functions are placed in the external memory. Some function in between are located in the external memory, but are loaded to the scratchpad on demand. This is done by a so called *page manager*. The decision which function will be placed in the scratchpad is

done by an optimisation algorithm to reduce the energy consumption. Since the *page manager* is implemented in software, the performance drawback will be higher than for a hardware-managed scratchpad like proposed in [Janapsatya et al., 2006b] or the D–ISP.

Udayakumaran et al. use a greedy heuristic for assigning code and data object to a dynamic scratchpad [Udayakumaran et al., 2006]. The proposed algorithm selects the program objects by a metric that estimates their usage frequency. The work is based on the dynamic data allocation for scratchpads published in [Udayakumaran and Barua, 2003]. The code is assigned to the scratchpad on the granularity of small functions that are generically created of selected parts of program structure (so called *outlining*). Udayakumaran et al. motivate their choice by the observation that smaller data memory objects typically yield better results when assigning them to the scratchpad[17], but on the other hand too small code objects pose long algorithm run-times and an increased amount of branches that have to be added to reconnect the code fragments (as is done by [Falk and Kleinsorge, 2009] for a static scratchpad). The authors show that a dynamic code and data scratchpad outperforms an optimal code and data allocation for a static scratchpad in terms of average case performance and energy consumption. To compare static and dynamic allocation Udayakumaran et al. enhance the optimal static algorithm for data scratchpads of Avissar et al. [2002] to also assign code to the scratchpad. Furthermore, in [Udayakumaran et al., 2006] a comparison of a software-based memory transfer and DMA is presented. The authors show that DMA provides a slightly better performance, because the proposed algorithm requires the transfer of only a low amount of data and thus the slower software-based transfer has only a small impact. Also the dynamic scratchpad is compared to direct mapped caches with the result that both have in average a comparable performance. Udayakumaran et al. state that due to the proposed scratchpad assignment the latency of every memory access is known a priori and thus the memory system has a predictable behaviour. Also it is known from Wehmeyer and Marwedel [2004; 2005] that an assignment algorithm that is not using the WCET in the cost metric can reduce the WCET estimate of the system and is even able to outperform a cache.

The approach of Puaut and Pais [2006; 2007] that is capable of assigning instructions to lockable caches and dynamic scratchpads by using a WCET-aware metric is already described in Section 2.3.1. To load the selected instructions into the scratchpad (respectively the locked cache) a copy function is used that will be invoked on a scratchpad reload point. Since insertion of the reload points that activate the copy function must not change the layout of the code, in [Puaut and Pais, 2006] two possible solutions are proposed: either debug registers are used to raise an exception once a reload point is reached or at any possible reload point in the code (which are not numerous, according to the authors) a placeholder for a call to the copy function is inserted before the assignment algorithm is started.

A rather uncommon approach of a software-managed scratchpad is proposed by Jacob. A special TLB manages the content of a scratchpad on the granularity of pages [Jacob, 1999]. It distinguishes if a memory access is routed to the main memory or to the scratchpad. The mapped memory regions can be adjusted during run-time by copying the new content into the scratchpad and setting the TLB entries properly.

Cong et al. propose a hardware extension to a cache that allows it to use a fraction of it as a scratchpad, the so called *adaptive hybrid cache* (*AH-Cache*). Therefore, a hardware lookup logic is proposed that distinguishes, if an access maps the scratchpad's address space or is in the cache. The scratchpad consists of multiple cache blocks, while the cache blocks that are not assigned to the scratchpad can be still used as cache [Cong et al., 2011]. A software interface allows to apply any content selection algorithm for the *AH-Cache*. The hybrid cache is proposed

[17] An observation that can also be confirmed for WCET estimates and static instruction scratchpads (see Appendix A).

to reduce energy consumption of a cache, but it could be also used for real-time systems as any other software-managed scratchpad.

2.3.4 Hardware-Managed Scratchpads

A hardware-managed scratchpad is similar to a software-managed, with the difference that instead of copying the content by software routines specialised hardware controllers are used. Due to the use of a hardware controller for the content management it is also possible that hardware-managed scratchpads allow the transparent addressing of their content, i.e. the address space of the scratchpad is hidden from the application. This is in contrast to the classification of caches and scratchpads of Jacob et al., that distinguished between *self-managed scratchpads* with their own address space and *transparent caches* [Jacob et al., 2007]. For the classification within this work the method of how the instructions or data is copied into the local scratchpad determines whether a scratchpad is classified as software- or hardware-managed[18]. For example the scratchpad proposed by Vander Aa et al. [2003; 2005] hides its address space, as also the D–ISP (see Section 3.2) does. Whereas, the scratchpad of Janapsatya et al. [2006a; 2006b] is directly addressed, i.e. the application needs to jump into the scratchpad's address space to execute the mapped code. Non-transparent addressing is more complex from the application side of view, because it requires additional jumps or altered call/jump addresses in the application. On the other hand transparently addressed hardware-managed scratchpads require an additional address mapping mechanism, as described by Whitham and Audsley [2009] for a data scratchpad. Independently of organisation and implementation of the scratchpad, the algorithms for selecting the scratchpad content proposed for software-managed scratchpads can also be applied to hardware-managed scratchpads and vice versa.

Vander Aa et al. propose a scratchpad memory, that stores code to reduce the overall energy consumption of the system. It is designed to support an existing instruction cache by being used to serve only the instructions of selected loop bodies [Vander Aa et al., 2003; Vander Aa et al., 2005]. The so called loop buffer is enabled by a special instruction, which defines the borders of the loop body that is to be mapped to the scratchpad. The content of the loop buffer is managed by a local controller, that dynamically loads code into the buffer, if the start address of the loop body defined in the enable instruction does not match the start address of loop body that is already stored in the buffer. The loop body is loaded from the instruction cache into the scratchpad during executing the first iteration of the loop. For any further iteration the instruction cache is disabled. Since the loop buffer is smaller and does not need a power consuming cache lookup, the power consumption of the system can be reduced. The loop buffer is disabled and the cache is turned on again, when the program leaves the address range of the mapped loop body. To select which loops should be copied to the buffer the authors propose a mapping heuristic. Furthermore, Vander Aa et al. estimate the optimal scratchpad size for a given set of benchmarks.

Janapsatya et al. use a so called *concomitance* metric to assign basic blocks to a scratchpad that are executed in temporal proximity [Janapsatya et al., 2006a; Janapsatya et al., 2006b]. The proposed algorithm also allows to identify basic blocks that are known to be executed with temporal distance and thus can be assigned to the same scratchpad region. The authors show that their algorithm reduces the overall energy consumption and increases performance in comparison to a standard instruction cache. A scratchpad controller [Janapsatya et al., 2004; Janapsatya et al., 2006b] takes care of loading the selected code from the off-chip memory into the scratchpad. It is activated by so called *Scratchpad Management Instructions*. On execution

[18]The term cache is used only for the classic cache, which features the content management on the granularity of a cache line and has for each cache line an associated tag memory entry.

of these instructions the processor is stalled and the scratchpad controller is in charge to load the requested code block. Within the controller a *Basic Block Table* stores the address of the code in the off-chip memory, the target address in the scratchpad, and the size of each mappable code block. By searching the table for the address that is encoded in the *Scratchpad Management Instruction* the controller identifies target and size of the code block and initiates the copy process. When the controller is finished, the processor continues and may jump into the scratchpad address space to execute the buffered code. The content of the *Basic Block Table* is filled a priori by the proposed scratchpad assignment algorithm. The scratchpad controller has no awareness about the content of the scratchpad and thus code will be loaded on request independently of the chance that it is already in the scratchpad. This is in contrast to the content management of the D–ISP. Anyhow, the absence of content awareness leads to a low hardware overhead of the scratchpad controller [Janapsatya et al., 2004; Janapsatya et al., 2006b], but it might result in an overhead in terms of performance and energy consumption or otherwise in a higher complexity of the scratchpad assignment algorithm. In [Janapsatya et al., 2004] also a scratchpad allocation algorithm is proposed that loads the highly utilised code parts into the scratchpad, but Janapsatya et al. show that the algorithm based on the *concomitance* metric is superior in terms of lower energy consumption and higher performance [Janapsatya et al., 2006a].

In [Lee et al., 1999] a small loop cache is proposed. It uses a hardware controller to detect a special branch instruction that activates the loop cache and also determines how many subsequent fetches should be buffered. The loop cache is active to serve the fetch requests until the mapped loop is left. Lee et al. show that this small buffer is able to significantly reduce the number of fetch access to the next level memory.

2.4 Data Memories in Real-Time Systems

2.4.1 Caches

Beside instruction cache analysis also a lot of research regarding data cache analysis is done, e.g. for direct mapped and LRU caches [Li et al., 1996; White et al., 1997; Ferdinand and Wilhelm, 1998; Huynh et al., 2011], multi-level caches [Lesage et al., 2009], and also unified L2 caches [Chattopadhyay and Roychoudhury, 2009]. A main difficulty of data cache analysis is that one load/store instruction can access multiple memory addresses. Furthermore, the addresses that can be accessed by a load/store instruction have to be determined by the analysis. Such memory address analyses are described for example in [Kim et al., 1996; Ferdinand and Wilhelm, 1998]. The preciseness of the memory address analysis directly affects the cache analysis, because if one memory address cannot be determined, the whole cache state could be corrupted [Reineke et al., 2007].

2.4.2 Scratchpads

Panda et al. provide a survey on memory optimisations for embedded systems that also covers the use of data scratchpads [Panda et al., 2001]. Furthermore, Panda et al. present a memory partitioning algorithm, that divides the on-chip memory into cache and scratchpad, to increase the overall performance of the system [Panda et al., 1997; Panda et al., 1999].

Udayakumaran and Barua propose an algorithm for the dynamic allocation of global and stack variables to a scratchpad. The data load and eviction can be performed by software or DMA transfers [Udayakumaran and Barua, 2003]. Udayakumaran and Barua compare their algorithm to an optimal static assignment algorithm, which was earlier proposed in [Avissar et al., 2002]. It is shown that the dynamic assignment clearly outperforms the static assignment algorithm in

terms of average case performance. In [Udayakumaran et al., 2006] it is shown that the proposed dynamic scratchpad (that is enhanced to also contain code) can compete with direct mapped caches.

Since a lot of memory accesses are directed to the stack, a special stack scratchpad is proposed by Park et al. The approach uses a sliding address map for the scratchpad to always follow the stack pointer [Park et al., 2007]. If the stack grows beyond the capacity of the scratchpad, a part of the stack is copied to external memory. On shrinking stack size the swapped data is dynamically reloaded. Thus, every stack access is directed to the scratchpad. Park et al. were able to show that their approach better exploits the locality of the stack than a cache, which results in a reduced energy consumption and increases system performance.

Other approaches use a scratchpad to improve the average case performance of the embedded system by allocating the frequently used arrays or the most accessed parts of the arrays, like the work of Kandemir et al. or Li et al. In [Kandemir et al., 2001] a software-managed scratchpad that dynamically loads frequently accessed parts of arrays is proposed. A graph colouring algorithm for scratchpad allocation of arrays is proposed in [Li et al., 2005].

Poletti et al. use a custom DMA controller to copy data between a scratchpad and the main memory [Poletti et al., 2004]. The DMA transfers are initiated by software using an also proposed high level API. A comparison of memory transfers by software and by the proposed DMA controller shows that the DMA controller allows both a lower energy consumption and an improved average case performance. This observation is not in common with the study of Udayakumaran and Barua [2003], that results only in a small overhead of software controlled memory transfers. Also a subsequent study by Udayakumaran et al. [2006], that additionally maps instructions to the scratchpad, results in a decent performance gain for a system using DMA transfers. The difference of the results of these studies may be caused by the fact that Poletti et al. use a first level cache that was not bypassed when copying the data by a software routine.

In [Suhendra et al., 2005] a WCET-sensitive scratchpad allocation for data is proposed. The authors present an ILP formulation that finds a data allocation while taking changes on the WCP caused by assigning data objects to the faster scratchpad into account. The proposed solution delivers the optimal data object assignment to the scratchpad while minimizing the WCET of the application. Suhendra et al. further improve the WCET calculation and the assignment algorithm of the data objects by taking infeasible paths into account. The infeasible paths are detected by a data flow analysis [Suhendra et al., 2006]. Falk and Lokuciejewski slightly enhance the work of Suhendra et al. and integrate it into their compiler framework [Falk and Lokuciejewski, 2010].

A dynamic data scratchpad allocation algorithm for real-time systems is proposed by Deverge and Puaut. To find the optimal assignment the data accesses that occur along the WCP are analysed. Then an ILP is used to assign the most beneficial data memory objects. In [Deverge and Puaut, 2007b] changes of the WCP, that are caused by assigning data to the scratchpad, are addressed by tuning the scratchpad allocation with an iterative algorithm. For evaluation Deverge and Puaut modified the algorithm of Avissar et al. [2002] to find an optimal static memory allocation to compare with. It is shown in [Deverge and Puaut, 2007b] that the proposed dynamic allocation provides lower WCET estimates than the static allocation, especially for small scratchpad sizes.

Whitham and Audsley propose a *scratchpad memory management unit* (SMMU) for a data scratchpad in hard real-time systems. The SMMU hides the address space of the scratchpad and is used to copy data from/to the scratchpad (like a DMA controller) [Whitham and Audsley, 2009]. The SMMU checks if memory requests access any of the mapped regions and routes the requests either to the correct position in the scratchpad or to the external memory. The

content management of the scratchpad is done explicitly by *open* and *close* operations. *Open* provides the source address in the external memory, the target address in the data scratchpad, and the size of the memory region to map. The SMMU is responsible to copy the requested data into the scratchpad. During data transfers the SMMU also checks whether the data is already mapped in the scratchpad. If so, it copies the data from the mapped scratchpad region to the new scratchpad address instead of loading it from the external memory. This is important, because the data in the scratchpad might have been altered by the application since their load. *Close* copies the data back to the external memory. If a part of the data is still mapped to the scratchpad by another mapping, it is copied back to the associated scratchpad address. Since the timing of *open* and *close* is predictable, they do not hinder a precise WCET analysis. In fact the WCET analysis is aware of when which data region is mapped to the scratchpad and thus precise WCET estimations can be provided. So Whitham and Audsley show that their proposed data scratchpad with the SMMU outperforms a common data cache, when considering the worst-case behaviour. The closer look on the implementation of the SMMU reveals that its internal timing characteristics strictly depends on the number of mappings it can maintain. An observation that is consistent with the evaluation of the D–ISP provided in Section 6.1. Because the SMMU is on the critical path of the data memory access and on every memory access a lookup is triggered to check if the requested address is mapped to scratchpad or not, also the maximum clock frequency of the host processor is affected [Whitham and Audsley, 2009]. This is in contrast to the D–ISP for which the lookup for the correct mapping is performed only on activating the mapping (by call or return). Hence, for the D–ISP the time (and energy) consuming lookup is intentionally taken off the critical path of instruction fetch. Only the address translation has to be performed by the D–ISP on every memory access, which has also to be done by the SMMU (after the correct mapping was found). The proposed SMMU is developed by Whitham and Audsley for data scratchpads, but it could also be used to transparently cache selected code parts. The restriction to use the SMMU for instructions only would also ease its implementation, because instructions cannot be changed by the application and thus do not need to be copied back to external memory on unmapping a memory region.

2.5 Off-Chip Memories in Real-Time Systems

DRAMs are commonly used as off-chip memories in embedded systems. Unfortunately their timing behaviour is more complex to model than for an SRAM. This is caused for example by the fact that the memory cells need to be frequently refreshed and the memory access time depend on various parameters and prior memory accesses. For a more precise description of DRAMs, their variants, organisation, and timing refer to [Jacob et al., 2007].

During the memory refresh each memory row has to be accessed, i.e. the memory row is read and written back immediately. Because the off-chip DRAM is not as small as on-chip memories, DRAMs are typically operated with a lower frequency than the processor, and the memory refresh has highest priority in the memory controller, refreshes block the memory for a large number of processor cycles. This behaviour is critical for real-time systems, since it is not known when the DRAM is refreshed (from application point of view they appear to be asynchronous) and the execution of the application can be stalled until the refresh is completed. A way to mitigate this effect is a distributed refresh that accesses only one row per time, but needs to be triggered more frequently. In this case the maximum delay one memory access can suffer by a refresh is reduced.

Park and Shaw examined that the memory refresh introduces non-determinism into the timing of a system and can have a noticeable effect on the worst-case performance of the processor. To

address the influence of the refresh in the WCET analysis the estimate is increased by the maximal observed overhead caused by the periodic refreshes [Park and Shaw, 1991]. Atanassov and Puschner propose a similar method to account the impact of DRAM refreshes in the WCET estimate by adding the delay of the maximum number of refreshes that can occur during the estimated WCET. The authors further show in [Atanassov and Puschner, 2001] that the DRAM refresh can even affect the WCP of the application, such that a WCET analysis without taking the refreshes into account might determine the wrong WCP. Anyhow, it is stated by Atanassov and Puschner that the computed WCET estimate will still be a valid upper bound of real system performance. In [Bhat and Mueller, 2010] it is revealed that the previous approaches only hold for non-preemptively scheduled systems, because in preemptive systems the task's execution can be interrupted, which leads to an asynchronous occurrence of the memory refresh. To address the refresh in such systems Bhat and Mueller propose to assign the DRAM refresh to a periodically scheduled task. Thereby, a schedulability analysis of the whole system can assure that the deadlines of all other tasks are met, without the need to consider the DRAM refresh in the WCET analysis of the application tasks.

Akesson et al. [2007] propose a predictable SDRAM[19] controller for real-time systems. The SDRAM controller guarantees a minimum bandwidth and maximum latency for memory accesses. Therefore, the memory accesses are encapsulated in access groups that provide a maximum memory efficiency and eliminate interferences between memory accesses, i.e. the timing of a memory access is not affected by any prior accesses. A dynamic arbitration scheme for the memory access groups ensures that the bandwidth and latency guarantees are met. In [Akesson et al., 2007] also an implementation of the proposed controller is presented. Paolieri et al. describe a real-time capable DRAM controller for a multicore. A round-robin arbitration of the memory accesses upper bounds the interferences of different hard real-time tasks. Hence, the maximum delay that one hard real-time task can suffer when accessing the off-chip memory depends only on the number of hard real-time tasks in the system [Paolieri et al., 2009a]. Furthermore, non hard real-time tasks are served by the proposed memory controller in a best effort manner. To cope with the unpredictability of the DRAM refreshes Paolieri et al. propose to synchronise the DRAM refresh with the application start, such that the impact of the refresh will be deterministic.

Reineke et al. propose to divide the address space of the DRAM into private partitions according to the DRAM's organisation structure. Thereby, the authors are able to isolate the memory accesses to the different partitions [Reineke et al., 2011]. The interleaving of the accesses in a time-triggered fashion furthermore exploits the parallelism of the banks and allows a higher memory bandwidth. Reineke et al. also propose a refresh scheme that is capable of deferring a refresh such that refreshes do not affect the latency of a single memory access and are predictable during DMA transfers.

It is also possible to model the behaviour of a DRAM controller during timing analysis of the system. For example Bourgade et al. propose an analysis of a memory hierarchy including a DRAM controller with an open page policy, i.e. after access the row is kept in the row buffer and is not automatically precharged. By predicting the content of the row buffer, the authors were able to show improvements of the WCET estimate [Bourgade et al., 2008].

2.6 Worst Case Execution Time Analysis

For schedulability analysis of tasks in hard real-time systems, their maximum execution time has to be determined. Otherwise no valid and safe schedule, in which all deadlines of the tasks are guaranteed to be met, can be created. A worst-case execution time (WCET) analysis is used to

[19] SDRAM – Synchronous DRAM

estimate the maximum execution time of a task. Since the timing analysis of the system and the task itself is not trivial, WCET analyses usually estimate only a safe upper bound and not the actual WCET [Wilhelm et al., 2008]. The tightness of this estimate to the real (typically unknown) WCET is a measure for the quality of the analysis.

In literature mainly two different approaches of WCET analysis are distinguished: the *static* and the *measurement-based* analysis. The static WCET analysis models the task and the system on which the task is executed, to calculate the WCET estimate. The quality of the estimate obtained by a static WCET analysis depends on the quality of the used model. To create a good system model all features of the processor, the memory subsystem, and other components that influence the timing of the system have to be taken into account. Measurement-based WCET analyses do not rely on a model of a system, they measure the timing of the system by executing the task itself. The drawback of measurement-based WCET analysis is that the quality of the obtained estimate strictly depends on the coverage of the measures regarding the program paths, the input set, and the initial hardware states. Those two analysis approaches will be discussed in the following.

2.6.1 Static WCET Analysis

Engblom et al. [2003] divides the static WCET analysis into three different steps: *program flow analysis*, *low-level analysis*, and *WCET calculation*. Furthermore, Engblom et al. discuss and compare a wide range of related work. Also Wilhelm et al. [2008] provide, more recently, a good overview on WCET analysis techniques and tools.

Program Flow Analysis

The program flow analysis determines the possible control flows of the application to allow the determination of the application's worst-case execution time critical path (WCP) at a later analysis step. Therefore, it is necessary to determine for example the maximum iteration count of loops (loop bound), the context of function calls, paths that are dependent on the same condition, and infeasible paths.

The program analysis first builds a representation of the application's control flow, which is typically a control flow graph (CFG). The complexity of creation of a CFG depends on the stage at which the analysis is performed. If the analysis is done within the compiler, the complete program information is at hand. A post-link analysis has to regain a lot of information [Theiling, 2000] and further also additional hints (like jump targets of indirect jumps) have to be provided to build the complete CFG.

To determine the possible control flows of a program the bound of loops need to be known. There are two possibilities for passing the loop bounds to the program flow analysis. The application programmer can provide this information to the analysis, e.g. by instrumenting the source code or creating an additional application profile (like *flow facts* used in [Ballabriga et al., 2011]). Another possibility is to determine the loop bounds automatically and provide them to the analysis. For further information regarding loop bound determination for WCET analysis refer e.g. to [Healy et al., 2000; Ermedahl et al., 2007; Cullmann and Martin, 2007; de Michiel et al., 2008]. Also the consideration of infeasible or conditional control flow paths is important for the computation of a tight WCET estimate. Therefore, different approaches for an automatic detection of such paths are proposed, e.g. in [Healy and Whalley, 2002; Gustafsson et al., 2006]. The program flow within a function may depend on the context in which the function is called. To allow a context sensitive interprocedural analysis different approaches as the *call string* approach [Nielson et al., 1999] or *VIVU (virtual inlining virtual unrolling)* [Martin et al., 1998]

are proposed. Beside detecting the program flow information it is also necessary to efficiently represent it for later stages of WCET analysis. For example in [Engblom and Ermedahl, 2000] a method is proposed that converts the description of complex control flows into linear constraints such that they can be used for WCET calculation.

Low-Level Analysis

The low-level analysis considers the behaviour and timing of the system, that executes the application. To allow a correct and precise low-level analysis the application needs to be available as linked binary file, because several system features like pipeline or cache depend on the address of data and instructions in the physical memory [Heckmann et al., 2003]. Therefore, the low-level analysis has to be performed within the compiler [Falk and Lokuciejewski, 2010] or by a post-link analysis tool [Ballabriga et al., 2011].

The analysis of the processor pipeline is one of the major topics for the low-level analysis. Analysis methods are proposed for very simple pipelines [Zhang et al., 1993; Lim et al., 1995], pipelines supporting multi-cycle instructions [Hur et al., 1995], superscalar in-order pipelines [Choi et al., 1994; Lim et al., 1998], and also out-of-order pipelines [Li et al., 2004; Li et al., 2006; Rochange and Sainrat, 2009; Barre et al., 2006]. The complexity of analysing out-of-order pipelines is that timing anomalies [Lundqvist and Stenström, 1999; Wenzel et al., 2005b] may occur, i.e. the local worst case of an event results not necessarily in the global worst case. For example Lundqvist and Stenström show that a cache hit can pose a worse global timing than a cache miss. Also the effect of timing anomalies might not be bounded and so called domino effects may occur. For further work on pipeline analysis refer also to [Engblom, 2002] and [Thesing, 2004].

Beside the pipeline also speculative features of the processor like branch predictors need to be considered in the low-level analysis, e.g. as done by [Colin and Puaut, 2000; Burguiere and Rochange, 2005; Burguiere and Rochange, 2007], or they have to be disabled to provide safe analysis results. The memory system that can contain caches, scratchpads, and off-chip memory needs special attention by low-level analysis, since the performance and timing of a processor is strongly influenced by the memory subsystem. Because the dynamic content management of caches is not easy to analyse, in embedded real-time systems also static or software-managed scratchpads are used. The major techniques of memory analysis were already discussed in previous sections of this chapter. The common analysis methods of caches and static scratchpads are revised in detail in Chapter 4. Also within Chapter 4 an analysis of the function-based D–ISP is proposed.

The analysis of the system's components can either be done separately within different analysis steps or by an integrated analysis. Heckmann et al. point out that there can be interferences between different processor components like cache and pipeline that cause mutual dependencies of their analyses and thus hinder the modular approach. To consider these interferences in a modular analysis framework the interferences of other components have to be upper bounded for each analysed component. But unfortunately, such approach might overestimate the possible interferences leading to very pessimistic WCET estimates. On the other hand an integrated and thus preciser analysis tends to be very complex [Heckmann et al., 2003]. In [Li et al., 2004] an instruction cache and branch analysis is integrated into the analysis of an out-of-order pipeline. This integration is necessary for the out-of-order pipeline, since e.g. for an instruction cache miss no fixed penalty can be determined due to the possibility of timing anomalies [Lundqvist and Stenström, 1999]. Also Lim et al. [1994; 1995] and Healy et al. [1995; 1999] integrate the cache and pipeline analysis and thereby are able reach tighter WCET estimates than a separated analysis.

WCET Calculation

Finally the WCET calculation combines the results from program flow and low-level analysis to determine the WCP of the application. Once the WCP is found the WCET estimate can be calculated. Three different approaches for WCET calculation are distinguished [Engblom et al., 2003]: *path-based*, *tree-based* and *IPET-based*. The path-based approach searches among all possible paths for the one with the maximum execution time. Whereas, the tree-based approach traverses the tree representation of the program from bottom up to find the path with the maximum execution time. For related work regarding path-based and tree-based approaches refer for example to [Stappert et al., 2001] and [Colin and Puaut, 2001] respectively.

Instead of traversing all possible paths of the program to find the WCP, the IPET approach [Li and Malik, 1995] expresses the program flow and the results from the low-level analysis as a set of linear constraints and an objective function. This representation is an ILP for which a solver can be used to find the solution with the maximal value of the objective function that determines the WCET estimate and the WCP. Puschner and Schedl show how the search for the WCP in the CFG can be formulated as a maximum cost circulation problem of a directed graph [Puschner and Schedl, 1997] that can be transferred into a similar ILP representation.

The complexity of the ILP in the IPET approach is a problem, because the run-time of a ILP solver increases super-linear with increasing the number of variables and constraints of the problem. Therefore, attends were made to reduce the complexity of the ILP solving by partitioning the problem into subproblems. Ballabriga and Cassé [2008b] split the control flow of the application into partitions by identifying *single-entry single-exit regions* (see [Johnson et al., 1994]). For these partitions the WCET can be calculated independently. To estimate the overall WCET the partitions are recombined and the application's WCET is calculated. By partitioning the ILP the run-time of the WCET calculation is noticeably reduced. Ballabriga and Cassé state that when the architectural constraints are taken into account, the splitting of the control flow has no effect on the precision of the calculated WCET. A slightly different approach is proposed by Ermedahl et al. that uses so called *fact clusters* for splitting the applications control flow [Ermedahl et al., 2005]. Using the flow information of the program, separable control flows can be identified and clustered. By respecting the flow facts, the WCET of the fact clusters can be calculated independently. To obtain the program's WCET estimate the fact clusters are merged from bottom-up and so the complete control flow of the application is reassembled.

As there are dependencies between the low-level analysis of different processor components, also dependencies between WCET path analysis (i.e. the search for and computation of the WCP) and low-level analysis are present. Again it is possible to integrate both steps or to use a separated analysis. For example Lundqvist and Stenström integrate the low-level timing analysis with the path analysis to provide tight WCET estimations [Lundqvist and Stenström, 1998]. On the other hand Theiling et al. present a separated but also accurate path and timing analysis [Theiling et al., 2000].

2.6.2 Measurement-Based WCET Analysis

Instead of modelling the hardware of the system a measurement-based WCET analysis executes the application on the real hardware or a simulator [Wilhelm et al., 2008]. To obtain the WCP and the WCET estimate of the application either end-to-end measurements of the whole application are performed or parts of the code are measured independently and the resulting path and execution time information are combined to build the WCP. Guarantees of the WCET estimates can only be provided for the set of covered input sets and initial hardware states. Wilhelm et al. states that due to this and the possibility of timing anomalies in complex processors, safe WCET

estimations can be obtained only for simple architectures using the measurement-based WCET analysis approach.

Bernat et al. combine measurement-based and analytical approaches to estimate maximum execution times that are close to the real WCET, but the estimates are not definitely safe. Therefore, such probabilistic method is applicable for systems in which a deadline miss is tolerable with a given probability [Bernat et al., 2002]. The WCET estimate is created by the composition of timing characteristics of basic blocks that are obtained e.g. by measurements of the processor hardware or a simulator.

Wenzel et al. propose a hybrid WCET analysis approach that uses both measurement-based and static analysis methods which is described in [Kirner et al., 2004; Wenzel et al., 2005a; Wenzel et al., 2009]. During analysis subparts of the application are measured independently and later composed to estimate the WCET. To mitigate the problem of path and input data coverage an automatic test data generation is integrated in the proposed analysis framework. Bünte et al. [2011a; 2011b] discuss the input test data generation in more detail and propose a metric for the input test data sets to evaluate their quality.

In [Deverge and Puaut, 2007a] a method for safe WCET estimations using measurement-based analysis is suggested. Therefore, Deverge and Puaut eliminate the timing variance of complex processor architectures by disabling the sources of unpredictability that can cause different execution times for the same program path, e.g. by using cache locking or static branch predictors. The authors have the opinion that their approach can provide safe and tight WCET estimates, but this is not convincingly proven.

2.6.3 WCET Analysis Tools

In the following a short introduction of some WCET tools that are related to this work is given. Either the introduced tools use techniques for memory analysis that are applied or used for evaluations in the following chapters (like aiT for caches and WCC for static scratchpads) or they support memories similar to the proposed D–ISP (like WCA that features the analysis of the method cache) or the D–ISP itself (as RapiTime and OTAWA).

These tools represent only an outline of the available WCET tools from industry and academia. For further WCET tools, like Heptane [Colin and Puaut, 2001] or Chronos [Li et al., 2007], refer to the survey on WCET tools by Wilhelm et al. [2008] or the see the reports of the "WCET Tool Challenge"[20] e.g. [Holsti et al., 2008] and [von Hanxleden et al., 2011].

aiT

The WCET tool aiT is a commercial static WCET analysis tool by AbsInt[21]. As input for the WCET analysis it uses the application's binary executable and additionally user annotations, that e.g. contain the recursion depth of a function or loop bounds. The aiT tool supports various processors like for example the ARM7, Freescale MPC5553/MPC5554 [von Hanxleden et al., 2011], and Motorola PowerPC 755 [Heckmann et al., 2003].

In the first step aiT builds a control flow graph of the application. As a commercial tool aiT tries to reduce the required user annotations to perform an analysis of an application. For example it detects indirect jump targets which are needed to construct the control flow. A loop bound analysis allows the detection of the maximum iteration count of simple loops. To extend the number of detected loop bounds aiT also searches for common loop patterns that are generated by different code generators and compilers [Heckmann and Ferdinand, 2006]. Furthermore,

[20] WCET Tool Challenge – `www.mrtc.mdh.se/projects/WCC` [Online, last accessed on 3rd March 2012]
[21] AbsInt Angewandte Informatik GmbH – `www.absint.com/ait/` [Online, last accessed on 3rd March 2012]

in [von Hanxleden et al., 2011] a feature of the tool Astrée[22] is briefly described, that allows the extraction of loop bounds and pointer targets from C source code, which can be used as annotations by aiT. So the user interaction required to obtain tight WCET estimates can be further reduced. To improve analysis quality for loops and recursive functions aiT uses the VIVU (virtual inlining virtual unrolling) approach of Martin et al. [1998].

In aiT for the pipeline analysis abstract interpretation is used, as described by [Schneider and Ferdinand, 1999]. Thesing [2004] extends the abstract interpretation based pipeline analysis to more complex processors that feature e.g. out-of-order execution. Beside the pipeline and path analysis aiT also supports the analysis of caches [Ferdinand and Wilhelm, 1999; Heckmann et al., 2003] and branch predictors [Grund et al., 2009] by using abstract interpretation. Furthermore, aiT supports register value analysis [Heckmann and Ferdinand, 2006] delivering an interval of possible register values that are used to determine the addresses of cache accesses and that are fed into the loop bound analysis.

OTAWA

The OTAWA[23] toolbox [Ballabriga et al., 2011] is an open framework for WCET analysis that is intended to support a wide range of target architectures and analysis techniques. OTAWA uses the binary of the application (e.g. as ELF file) as input. A set of ISAs[24] like PowerPC, ARM, and TriCore are supported by OTAWA. The ISA is hidden by an abstraction from the rest of the analysis. Therefore, the GLISS tool [Ratsiambahotra et al., 2009] is used. It generates loader modules for the binaries for the different ISAs. So the representation of the loaded application in the analysis process is made independent of the specific ISA of the host processor.

To allow architecture specific analysis a hardware description is provided by the user. Due to the structure of OTAWA the different target architectures and analysis types can be combined easily. The interfaces between the analysis steps allow to extend the analysis or add new architectural or program analysis [Cassé and Sainrat, 2006]. OTAWA currently supports the following architecture specific analyses: pipeline analysis for super-scalar out-of-order pipelines [Barre et al., 2006; Rochange and Sainrat, 2009], the analysis of instruction caches using abstract interpretation [Ballabriga et al., 2008; Ballabriga and Cassé, 2008a], and the analysis of dynamic branch predictors [Burguiere and Rochange, 2005; Burguiere and Rochange, 2007].

Automatic loop bound detection is supported by the oRange tool [de Michiel et al., 2008; de Michiel et al., 2010]. It provides the analysis performed by OTAWA with the loop bound information derived from the application's C-code. The oRange tool is capable of determining context dependent loop bounds, e.g. for a loop for which the iteration count depends on the context in which it is executed.

For calculation of the WCET OTAWA uses the IPET approach and to solve the ILP the tool `lp_solve` [Berkelaar et al., 2008] is employed. OTAWA supports the optimisation of the analysis run-time by reducing the time the ILP solver needs to calculate the WCET. Therefore, Ballabriga and Cassé [2008b] partition the CFG into subgraphs for which the WCET can be calculated faster and then these subgraphs are combined to calculate the overall WCET estimate.

In addition to the features discussed above OTAWA was used within the MERASA project. Therefore, it was extended to support the 2-way super-scalar in-order pipeline of the CarCore processor as described by [Landet, 2008; Landet, 2009] and the whole MERASA architecture including memory hierarchy and bus as shown in [Ungerer et al., 2010]. Also the D–ISP was

[22] A verification tool from AbsInt. Refer to `www.absint.com/astree/` [Online, last accessed on 3rd March 2012].

[23] OTAWA (Open Tool for Adaptive WCET Analysis) is available under LGPL at `www.otawa.fr` [Online, last accessed on 3rd March 2012].

[24] ISA – Instruction Set Architecture

modelled in OTAWA using abstract interpretation. In contrast to the ISPTAP tool (proposed in Chapter 5), OTAWA supports only the FIFO replacement policy for the D–ISP and does not implement the VIVU approach [Martin et al., 1998], instead a persistence analysis[25] [Ballabriga and Cassé, 2008a] is performed. The impact of the D–ISP on the WCET is published in the Deliverable 3.3 of the MERASA project [MERASA, 2009b]. In Section 6.2 a broader evaluation of the D–ISP including a comparison of different replacement policies and competing memories is presented.

RapiTime

RapiTime (see [RapiTime] or [Wilhelm et al., 2008]) is a commercial WCET analysis tool[26]. As a measurement-based analysis tool RapiTime traces the execution time of executed paths of the application under observation. Hence, the application has to be executed on the target system with a sufficient number of input sets. To allow meaningful results the complete code of the application needs to be covered by the chosen input sets. Furthermore, an analysis of the application's structure combines the execution cost of different measured program paths, such that RapiTime is able to estimate the timing of program paths that were not traced during the measurement. Therefore, RapiTime can find the WCET-critical path of the application without observing it.

By tracing the execution paths and visualising the impact of the execution cost within the application's code, the "worst-case hot spots" [RapiTime] can be shown to the developer. Hence, RapiTime provides good opportunities to optimize applications from the WCET perspective. RapiTime also offers mechanisms to ease the debugging of the system [RapiTime].

RapiTime was used in the MERASA project as measurement-based WCET analysis tool for the analysis of the multicore architecture as shown in [Paolieri et al., 2009b]. During the project RapiTime was also extended to support the D–ISP. An evaluation of multiple benchmarks using RapiTime and D–ISP is presented in [Ungerer et al., 2010]. Furthermore, in [Gerdes et al., 2011] a case study for the MERASA processor including the D–ISP with an analysis performed by RapiTime is described.

WCA for the JOP

The WCET analysis tool WCA[27] proposed by [Schoeberl et al., 2010] is designed for the JOP (Java Optimized Processor) [Schoeberl, 2008]. The JOP implements the Java virtual machine and executes the Java bytecodes by splitting them into one or more microcodes. To obtain the timing of the Java bytecodes the pipeline of the JOP is modelled. Because of the fact that the Java virtual machine is a stack architecture, the JOP can feature a very simple four staged pipeline. The JOP pipeline does not need data forwarding or has any data dependencies between microcodes, such that the timing analysis is kept simple.

The WCET analysis of the JOP is based on the IPET approach[28] by [Puschner and Schedl, 1997]. It is possible in the WCA that either the WCET is calculated for each method separately and then the costs of the method is charged on its invocation or one single ILP is generated. The latter case is necessary if e.g. the cache content is to be taken into account. In addition

[25] The persistence analysis is to identify memory references that result in a miss at their first access, but for any further access they will result in a hit, e.g. a loop in which a memory access in the first iteration produces a miss, whereas for all other iterations the memory access results in a hit.

[26] See website for further information: `www.rapitasystems.com` [Online, last accessed on 3rd March 2012].

[27] The WCA tool is available under GPL at `www.jopdesign.com` [Online, last accessed on 3rd March 2012].

[28] In [Harmon et al., 2008] a tree-based WCET analysis for the JOP is proposed and compared to the IPET approach.

to the WCET analysis the WCA is capable to detect the bounds of loops in the application by a data flow loop bound analysis using abstract interpretation. To support virtual method calls the WCA features a 0-CFA analysis (see [Nielson et al., 1999]) that narrows down the possible the methods that might be invoked. Therefore, the analysis tracks the types of the objects on their creation, such that on virtual method invocation a set of potentially called methods can be extracted.

The WCA also supports an analysis of the method cache [Schoeberl, 2004]. The used approaches to model the cache are described in [Huber and Schoeberl, 2009; Schoeberl et al., 2010] and were already discussed in Section 2.3. For analysis of the method cache using model checking Uppaal is used. Uppaal [Larsen et al., 1997; Behrmann et al., 2004] is a tool for model checking and verification using timed automata, but it can also be used for estimating the system's WCET, as Metzner [2004] proposes.

WCC - WCET-aware C Compiler

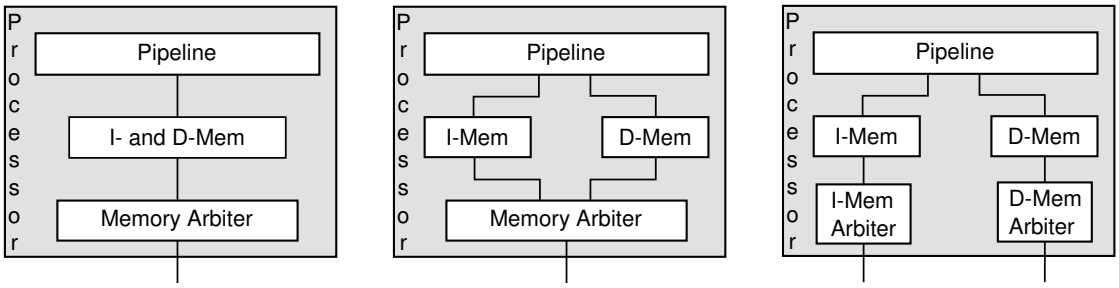
The WCET-aware C Compiler (WCC) described in [Falk et al., 2006; Falk and Lokuciejewski, 2010] supports the Infineon TriCore processor and integrates several compiler optimisations targeting the reduction of the WCET. Because the WCC can perform optimisations on different levels (C code representation and retargetable low-level intermediate representation), it has a broader view on the code structure than other WCET analysis tools, that (have to) work with the linked binary code. Hence, it can perform optimisations that exploit information that is usually lost in the compiled binary code.

The WCET analysis itself is outsourced by the WCC to the aiT tool [Heckmann and Ferdinand, 2006]. Therefore, aiT is integrated into the compiler framework, such that it can calculate the WCET and the WCP of the generated code during compilation. Thus, WCC can optimise the application regarding WCET during compilation and provide a WCET optimised binary file. Furthermore, WCET analysis results can be provided to the user. The WCC can make use of several architectural analysis features of aiT, like for example its cache analysis.

But also the WCET analysis can exploit the global view on the application that the compiler has to improve its WCET estimates. For example the compiler uses techniques for loop bound and address analysis at C code level that can be more precise than any analysis on binary code, because at the C code representation level more structural information of the program is available that can be provided to the WCET analysis without user interaction. Additionally, the WCC implements a source code level flow fact annotation, allowing to transfer flow facts given by the user or extracted by loop analysis to the aiT WCET analysis.

The WCC also optimises the application code at the low-level code representation. A reduction of the WCET estimate can be reached by a proper placement of code [Falk and Kleinsorge, 2009] and data [Falk and Lokuciejewski, 2010] to fast scratchpad memories. The implemented allocation technique is derived from [Suhendra et al., 2005], which is sensitive to changes of WCP, and calculates an optimal allocation for basic blocks and data objects. The approach presented in [Falk and Kleinsorge, 2009] is also described in Section 4.1, because it is used as competitor for the D–ISP in the evaluation of Chapter 6. Furthermore, the WCC provides compiler optimisations like procedure cloning [Lokuciejewski et al., 2008b], procedure positioning, and register allocation that allow the reduction of the WCET estimates. The procedure positioning is performed with respect to instruction cache effects and is proposed in [Lokuciejewski et al., 2008a]. Moreover, a WCET-aware register allocation reduces the number of spills[29] on the WCET path and thus the WCET estimate of the application can be also improved but with the cost of a larger code sizes [Falk and Lokuciejewski, 2010].

[29] Loading a variable from memory into a register before using it and storing it back to memory after usage.



(a) Shared first level memory and shared off-chip connection

(b) Separated first level memory and shared off-chip connection

(c) Separated first level memory and separated off-chip connection

Figure 2.2: Different memory hierarchy organisations of the first level memory and the off-chip memory connection

2.7 Memory Hierarchy Design and WCET Analysis

This section describes the influence of the memory hierarchy on the precision and complexity of the WCET analysis for the whole system. At the processor core level different memory hierarchy organisations can be distinguished, see Figure 2.2(a) to (c) for three common hierarchies. Each of them has a different impact to the quality of results and complexity of the WCET analysis.

A shared first level memory, as depicted in Figure 2.2(a), in which instructions and data are maintained by a single memory structure (e.g. a cache or a scratchpad) raises some problems for the WCET analysis. If the memory is not strictly partitioned[30] into data and instruction regions, both types of memory can change the memory content and thus interfere with each other. So the behaviour of one memory access type can directly affect the accesses of the other memory access type, e.g. by the eviction of data or instructions. Furthermore, pessimism in the analysis of the data memory may propagate to the instruction memory analysis: Consider the case in which an accessed data memory address cannot be predicted and an unknown address has to be assumed. Then to be safe the data access could access any (possible) address in the memory, causing the invalidation of the whole memory content without distinction of instructions or data. Naturally the propagation of imprecision is also possible vice versa, but usually the data memory analysis inherits more pessimism than the instruction memory analysis, because for an instruction fetch the memory address is always known. So this integration of first level data and instruction memory can easily lead to a high overestimation in the WCET estimate. Independently of the partitioning of the memory, instruction and data memory accesses affect each others timing, e.g. when a fetch and a data memory access occur at the same time. Hence, Wilhelm et al. [2009] suggest to use separated first level instruction and data memories.

The separation of the first level instruction and data memory as depicted in Figure 2.2(b), reduces the interference between both memory types, such that the content of both memories cannot be affected by each other. The source of interference in this scheme is the shared connection to higher memory levels that can be located off-chip, which is exemplified by the memory arbiter in the Figure 2.2. Basically every memory access that cannot be handled by the local memory can collide at the memory arbiter level with a similar access from the other memory. So in the worst case each memory access that leads to the higher memory hierarchy will be delayed by a data or instruction memory access, respectively. Since in a realistic scenario this will never happen, the obtained WCET is heavily overestimated. To tackle this overestimation

[30] In case of strict partitioning a "separated" first level memory can be assumed, as shown in Figure 2.2(b).

an analysis of the memory requests at the memory arbiter level is beneficial. But according to Heckmann et al. [2003] such analysis is quite complicated, because it has to integrate the pipeline timing, the timing and content of the local memories, and the memory arbiter itself. During the pipeline analysis it has to be determined when which fetch is requested and when which data memory access is triggered. The analysis of the local memory then has to check which of these memory accesses miss the local memory and are passed to the memory arbiter level. Depending on the microarchitecture, the pipeline can be stalled until the memory access is served. The analysis of the local memory has also to consider the internal timing of the miss handling and the organisation of the local memory (e.g. to obtain the address and the size of the cache line that is to be requested on a data/instruction miss). Finally, the analysis of the memory arbiter that models the timing of the next level memory (e.g. DRAM, flash, or L2 cache) determines, which memory accesses from the pipeline actually interfere with each other. So it can quantify the penalty of the occurring interferences for each memory request. The analysis at the memory arbiter level then provides the timing information of the memory accesses to the local memory analysis, which in its turn affects the pipeline analysis. Due to the inherent complexity of each analysis step, abstractions have to be made that result in a gain of pessimism, which propagates on integration of the analysis steps. So a balance of analysis complexity and preciseness has to be found. Nevertheless, an integrated analysis has the potential of preciser estimates, as e.g. shown by Lim et al. [1994; 1995], Healy et al. [1995; 1999], Heckmann et al. [2003], and Li et al. [2004].

The full separation of the instruction and data memory system as shown in Figure 2.2(c) is a good choice for the predictability of the system, because then no interferences of data and instruction memory access can occur. Thus, the WCET analysis does not need to take such effects into account to deliver precise results. Since the memory interferences are a cross-cutting issue in WCET analysis that require the orchestration of instruction memory, data memory, and pipeline analysis, the complexity of the WCET analysis is reduced when interferences can be eliminated by design. In result for such memory design a simpler timing analysis and tighter WCET estimates can be provided for the processor, when considering no higher level memories. On the other hand the separation of the memory hierarchy can affect the system's cost, e.g. by the two memory arbiters, the doubled number of off-chip connections (that might result in a higher pin count of the chip), and the separated next level memories. Furthermore, such memory hierarchy restricts the flexibility of the memory layout according to the partitioning of off-chip data and instruction memory.

Within this work a shared off-chip memory connection (refer to Figure 2.2(b)) is assumed, for which it will be shown that by the use of the proposed D–ISP the interferences at the off-chip connection are eliminated and the memory hierarchy becomes interference-free. Hence, from timing analysis point of view the memory hierarchy behaves like a strictly separated memory hierarchy (see Figure 2.2(c)). This allows a precise timing analysis of the system without the need to take any interference that require a costly integrated analysis into account.

Chapter 3

Dynamic Instruction Scratchpad

In this chapter the *Dynamic Instruction ScratchPad* (D–ISP) as a real-time capable instruction memory is proposed. The D–ISP features a dynamic content management like a cache with the difference that the content is maintained on the granularity of functions. The main benefits of the D–ISP are:

- The D–ISP ensures a predictable timing of the fetch requests, since all fetches are handled with minimum latency.
- The D–ISP employs a dynamic function-based content management that is completely implemented in hardware. Thus, in contrast to software-managed scratchpads the content management is performed with minimal timing overhead.
- The D–ISP eases a precise static WCET analysis by its design.
- The WCET estimate of a system can be reduced by the D–ISP compared to other common memories like scratchpads with fixed content or caches.

This chapter is structured as follows. An overview on the Dynamic Instruction Scratchpad is provided in Section 3.1. The Section 3.2 describes the D–ISP architecture in detail. The implementation of the proposed architecture in a simulator and hardware is discussed in Section 3.3.

The D–ISP architecture has been published in [Metzlaff et al., 2008; Metzlaff et al., 2011a] and was used within the MERASA project as predictable first level instruction memory [Ungerer et al., 2010].

3.1 Overview

The D–ISP is intended to be used as first level instruction memory for hard real-time architectures. As typical other first level memories it uses SRAM, which has a low and constant memory access latency. It can be used in a hierarchy with other memories like data caches, scratchpads, or second level caches.

The D–ISP is tailored to systems running application code that uses functions or similar structures that bind a certain functionality to a delimited part of the code, because it uses functions as entities that reside in the assigned scratchpad memory on the whole or not. The basic operation of the D–ISP is as follows: on function activation it is ensured by the D–ISP that the activated function is contained in the local memory. This is done by copying the function into the scratchpad, if it is not already hosted by the D–ISP. While function execution every fetch

request will be directed to the local scratchpad memory. This leads to a two-phased execution scheme in which instruction and data memory accesses to the next level memory are temporally separated:

1. **Execution phase**: While execution all fetches are handled by the D–ISP and no fetch request is delayed by a memory access to a memory beyond the D–ISP.
2. **Content management phase**: On call or return the execution is stalled to allow the D–ISP to copy the activated function into the scratchpad. This can be done partly in the slipstream of the usual call/return handling of the processor.

By the fact that every fetch request is handled by the D–ISP during execution, no instruction fetch will lead to any higher memory or shared resource. This reduces the pressure on the memory hierarchy. It also eliminates interferences of data memory accesses and instruction fetches directed to a shared resource. On call and return the D–ISP can operate in the slipstream of the call/return handling of the processes. The processor has to save respectively restore the context of the activated function, which usually takes several cycles. So the lookup time of a function is not critical for the D–ISP, since there is some time available to check the D–ISP content for the activated function while the processor is occupied with context management. This is in contrast to caches, where the hit/miss detection is on the timing critical fetch path. If the processor maintains a separated context memory like the Infineon TriCore [TriCore, 2007a], the D–ISP is also able to start function copying while the processor still saves or restores the context.

The function-based dynamic content management of the D–ISP is fully implemented in hardware. Thus it will outperform any software-based content managed scratchpads on loading and evicting functions. Furthermore, the software support that is necessary to use the D–ISP is minimized, since no memory mapping or assignment tools are needed. Tool support is only required to help the D–ISP detecting the sizes of the functions, such that it is able to load the whole function into the scratchpad.

From the WCET analysis point of view the D–ISP has advantages regarding the complexity of analysis and the tightness of the estimated WCET. By ensuring that every function is maintained in the D–ISP before its execution, the function execution will not be delayed by any fetch request. So the analysis does not have to consider different fetch latencies during function execution. Any penalties for filling the D–ISP can be assigned to call and return instructions, because only at this points the content of the D–ISP may change. Due to the fact that the D–ISP handles every fetch request equally, effects depending on instruction alignment or memory layout that may lead to a variable timing in cache memories (see e.g. [Bradford and Quong, 1999] or [Mezzetti et al., 2008]) cannot occur.

To connect instruction and data memory of the processor to the higher memory hierarchy either a shared or a separated off-chip connection is possible, as described in Section 2.7. By the two-phased execution scheme that enforces the temporal separation of data and instruction accesses, a physically shared off-chip memory connection behaves as a separated off-chip memory connection. So due to the temporal separation of data and instruction access, the timing analysis can assume that the instruction memory hierarchy is physically separated, even if it is shared. If an interference of both memories would be possible (as in common architectures), the timing analysis has to respect potential collisions. This can be done either by assuming an interference for every memory access and using the worst-case memory latency for the memory accesses, which would lead to a high overestimation of the WCET. Another solution to address this problem is an integrated analysis that can determine, which memory access may interfere with another one at the off-chip memory connection level and which access does not. See Section 2.7 for a precise description of the effects of the memory hierarchy on the WCET analysis.

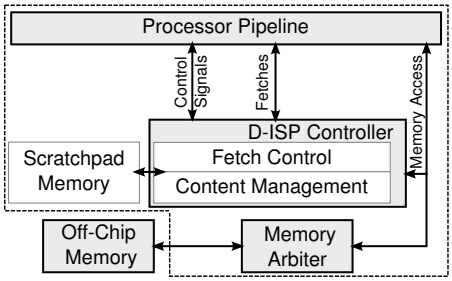


Figure 3.1: Overview of the D–ISP controller and integration into processor

If a D–ISP is used as instruction memory, the WCET analysis is able to employ a separated analysis of instruction and data memory at processor level. So the D–ISP allows a usage of a shared off-chip memory connection without any impact on analysis complexity and preciseness. This is also beneficial from the hardware point of view, since the redundancy of a separated off-chip memory connection can be saved. Also the memory controller can be kept simple, because no complicated arbitration is needed since instruction and data accesses are temporally separated. Thus the system can benefit from the advantages of a separated off-chip memory connection without paying its cost.

3.2 Architecture

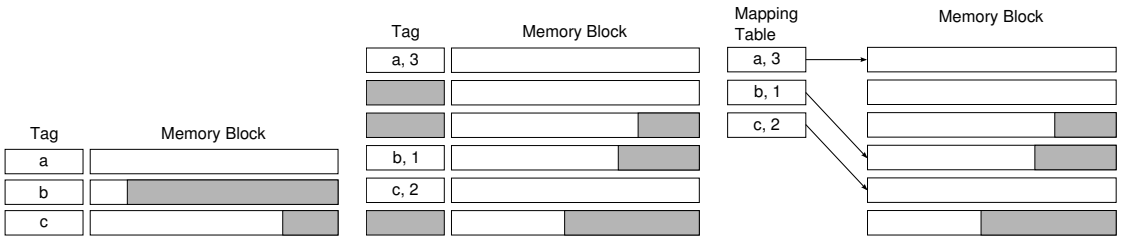
As the D–ISP manages its content during run-time a custom scratchpad controller, the *D–ISP controller*, is necessary to support the function-based operation. The D–ISP controller has two basic tasks, first the response of fetch requests and second the management of the scratchpad content, including content lookup, function load, and eviction. Therefore, it consists of two mainly independent parts: the *fetch control* and the *content management*. The structural overview of the D–ISP is shown in Figure 3.1: it consists of a controller and the scratchpad memory. In the scratchpad memory the instructions of the functions maintained by the D–ISP controller are stored. It is located near the D–ISP controller on the processor die and has its own separated address space.

3.2.1 Function Mapping

The D–ISP loads functions into the scratchpad memory, so it was to be aware of which function is in the scratchpad and which is not. Therefore, it has to keep information regarding all functions in the scratchpad. The so called mapping information contains the function address in the native address space[1] ($addr_N$), the address of the function in the scratchpad address space ($addr_D$), and the function's size. The function address in the native address space is used to correctly identify the function. The mapping information is needed to lookup functions on their activation and transform addresses from native to scratchpad address space during their execution.

To maintain the mapping information two possible alternatives can be distinguished: First the usage of a tag memory like used for caches, where each memory block is assigned to a memory tag. The second possibility is the usage of a mapping table in which each entry points to a block in the memory.

[1] The native address space is the global address space used for data and instructions.



(a) Tag-based organisation with one memory block per function

(b) Tag-based organisation with multiple memory blocks per function

(c) Mapping table-based organisation

Figure 3.2: Different organisations of the function mapping

Because functions do not have a union size, the tag memory approach has to be modified: Either the memory blocks have to be as large as the largest function and for smaller functions only a part of the block is used, as depicted in Figure 3.2(a). This approach is not desirable, because small functions will require the same memory amount as large ones, resulting in a very low memory utilisation. A better way is to use multiple blocks for one function as shown in Figure 3.2(b). Then multiple tag entries could be assigned for a single function, while only one is needed.

Using a mapping table instead of a tag memory, as depicted in Figure 3.2(c), only one entry per function is used that also holds the number of memory blocks used by the function. Each table entry points to a specific block in memory and so the function in the scratchpad can be accessed. The number of functions that can be stored in the scratchpad is limited to the number of mapping table entries and it is also bounded by the size of the scratchpad.

Regardless of the organisation of the mapping information the usage of memory blocks causes a loss of usable memory due to internal fragmentation. The fragmentation occurs if the function size is not an exact multiple of the memory block size. To reduce the fragmentation a finer granularity of memory blocks can be used[2], but this leads in for the tag-based organisation to a higher number of tag entries that are possibly unused. If a mapping table is used the number of entries is decoupled from the organisation of the memory, such that the fragmentation can be reduced, while the number of entries stays constant. From hardware point of view a large number of entries for function lookup leads either to a very high hardware effort or to a long lookup time. Thus, the number of tag entries and mapping table entries should be limited. This aspect of the implementation will be discussed in Section 6.1 in detail.

Due to the fact that functions have variable sizes and a mapping table allows a low fragmentation with a reasonable number of mapping table entries, the mapping table approach is used for the D–ISP. The mapping table contains an entry for each function that is contained in the scratchpad. If there is no mapping entry for a function, it is not stored in the scratchpad and has to be loaded on its activation. Therefore, the function mapping table can be used for the function lookup by the *content management*.

3.2.2 Fetch Control

A main part of the D–ISP controller is the *fetch control*, as depicted in Figure 3.3. The objective of the *fetch control* is the handling of incoming fetch request from the pipeline, by selecting the correct block in the scratchpad and delivering the requested fetch word back to the pipeline.

[2]This is not possible for the tag-based organisation with one memory blocks per function

Therefore, the *fetch control* needs to be aware of the current execution context represented by the currently active function. So the mapping information of the active function is needed, which is provided by the so called *context register*. Using the context register the *fetch control* is able to map every fetch request from native address space to the correct address in the scratchpad memory.

Equation (3.1) shows how the native address requested by the pipeline ($addr_N(x)$) is transferred into the scratchpad address ($addr_D(x)$) by the *fetch control*. The data hold in the context register is the function address in native address space (CR_N), the function address in scratchpad address space (CR_D), and the function size (CR_S). If a fetch request leaves the context of a function, which is not intended in normal operation, there is no valid address in the D–ISP and the fetch request is ignored by the *fetch control*. How this case will be handled when implementing the D–ISP in hardware is described in Section 3.3.3.

$$addr_D(x) = \begin{cases} addr_N(x) - CR_N + CR_D & \text{if } CR_N \leq addr_N(x) \leq CR_N + CR_S \\ invalid & \text{otherwise} \end{cases} \tag{3.1}$$

The mapping from native addresses to scratchpad addresses is rather simple, therefore the D–ISP will add no penalty on the critical path of instruction fetch and execution. Thus, fetches can be handled with minimum latency.

While the *content management* is busy, e.g. with loading a function into the scratchpad, the *fetch control* is deactivated. Then any fetch request is delayed until the *fetch control* is released by the *content management*. This is done to ensure that only valid data is delivered to the pipeline, because the *content management* changes the content of the scratchpad, the mapping table, and the context register. For example, when answering fetch requests and the context register is in an inconsistent state, wrong instructions could be executed. Also if the function is not completely loaded into the scratchpad, the *fetch control* has to be stalled, because otherwise the function is executed and data accesses (loads and stores) could interfere with memory accesses requested by the D–ISP to load the function. This would disrupt the constraint of the two-phased execution scheme, in which no data access conflicts with any fetch access, which is crucial for defining the timing upper bound for off-chip memory accesses.

3.2.3 Content Management

The main task of the *content management* is to ensure that the instructions for the incoming fetch requests can be handed by the *fetch control*. Therefore, the *content management* has to perform the following tasks:

- call and return detection and function identification
- function lookup
- maintenance of the mapping table
- function load and eviction

On function activation (via call or return) the *content management* has to check the content of the scratchpad for the activated function. Based on the function lookup the *content management* distinguishes two cases: either the function is already present in the scratchpad or not. Since the function activation may take time, the D–ISP controller has to stall the processor pipeline by delaying any fetch requests to the activated function. In case that the function is in the scratchpad, the *content management* can re-allow the execution of the function immediately.

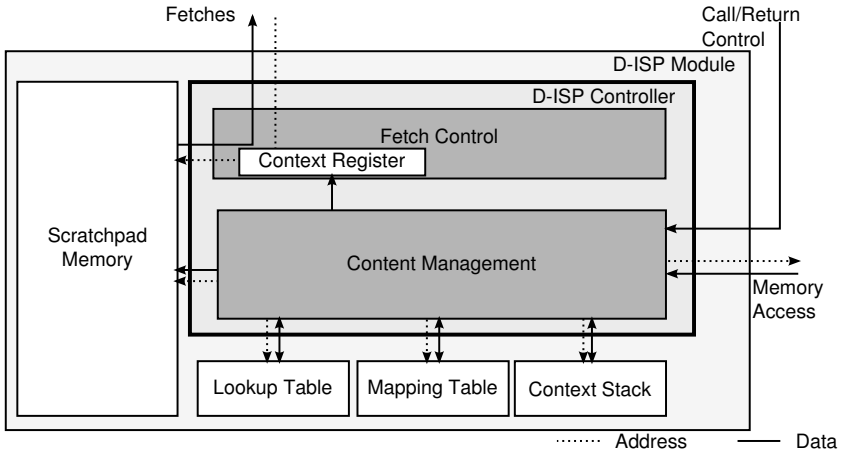


Figure 3.3: D–ISP controller including helper memories

Otherwise the function has to be loaded into the scratchpad. On completion of the function load the pipeline is released and the function is executed from the D–ISP.

The *content management* needs specific information of the maintained functions. The required data is stored in three additional helper memory structures: the *lookup table*, the *mapping table* and the *context stack*. This helper memories are shown in Figure 3.3. Also the *context register* is depicted, which is set by the *content management* and read by the *fetch control* to find the active function in the scratchpad memory.

Call and Return Detection and Function Address Determination

The *content management* is usually inactive during the execution of a function, except when a call or return is invoked. Then while the processor manages the context switch the D–ISP controller performs the preparation of the activated function to allow its execution from the scratchpad memory. So the operation of the *content management* can partly overlap with the context handling done by the processor.

A call or return will be detected either by additional signals e.g. introduced in the decode stage of the pipeline or the *content management* has to decode the instructions executed in the pipeline on its own. On call/return detection the *content management* has to obtain the address of the activated function to perform the function lookup.

For calls the identification of the invoked function is simple, because the function address is calculated in the pipeline. Using the result of the address calculation from the pipeline the D–ISP is able to handle every type of call the processor supports, including also indirect calls[3].

Function returns cannot be handled this easy by the *content management*. On return the pipeline restores the context of the caller function and reactivates it with the program counter pointing after the call to the returned function. There is no common way to obtain the address of the reactivated caller function by using internal pipeline processing. So either the caller function address has to be stored on the processors stack or the D–ISP *content management* has to maintain its own function call stack. The first option requires major changes to the context handling either in the pipeline or in the compiler. So to minimize the changes to the system architecture the *content management* features its own *context stack*, depicted in Figure 3.3. On

[3]For indirect calls the called function is not fixed at compile time, instead it will be determined at run-time.

call the address of the called function is pushed onto the context stack. On return the callee function address is deleted and the caller function address is on top of the context stack and can be obtained. Using the context stack the function that is reactivated on the return of a callee function can be identified without additional effort.

The detection of calls/returns and the determination of the function addresses requires changes to the host processor, which will be discussed in more detail in Section 3.3.

Function Lookup

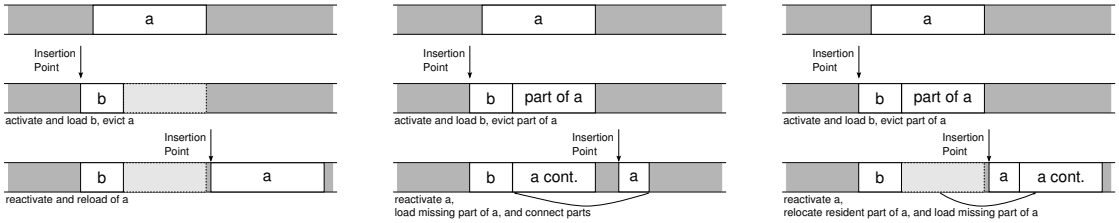
After obtaining the function address on call or return the D–ISP *content management* has to check the scratchpad content for the presence of the activated function. This can be done by checking all entries of the mapping table (see Figure 3.3) for the activated function. If the mapping table is accessed sequentially, the maximum search latency is commensurate to the size of the mapping table. Furthermore, the lookup time depends on the position of the address in the table. So the sequential lookup has two drawbacks: the long time it needs to find a function in the worst case and the uncertainty about the real lookup time. To reduce this uncertainty for a WCET estimation an analysis of the entries and their position in the mapping table could be performed.

To address the problem of a slow and unpredictable function lookup a special *lookup table* (see Figure 3.3) is proposed. This table assigns the mapping table position to the function address and allows to check multiple entries in parallel. If a function is found in the lookup table, the corresponding entry in the mapping table is accessed directly. This approach is comparable to the lookup done in an associative cache, where the cache tags are compared in parallel and on hit the right set is selected [Hennessy and Patterson, 2006]. As for a cache the number of parallel comparisons is crucial for the hardware effort and the critical path of the implementation of the D–ISP controller. To allow a low hardware effort and a large mapping table, the function lookup can be split into several cycles to reduce the number of parallel address comparisons. On the other hand this so called *multi-cycle lookup* increases the lookup time and also it introduces variance into the time a function lookup can take. In the Section 6.2.7 these aspects of the multi-cycle lookup are quantified in terms of hardware cost savings and WCET estimate increase.

Function Mapping Maintenance

If the function lookup finds the activated function, the D–ISP *content management* sets up the context register and releases the *fetch control* to start the execution of the function. Otherwise the function is not in the scratchpad and has to be loaded. Before function load the *content management* has to create an entry in the mapping table to bind the function address to the address in the scratchpad to which it will be copied and to store the function's size. Also a linked lookup table entry is created to find the mapped function on a later reactivation.

To load a function completely into the scratchpad the D–ISP *content management* has to know the function's size. There are three possibilities for the *content management* to obtain the function size: a function length table, instrumentation of the functions, and loading consecutive instructions into the scratchpad until the end of the function is reached. Using a function length table the D–ISP has to get the size of the function from the table on a miss. Depending on the location of the table the access costs will differ. A function length table located near the D–ISP controller would be fast, but this memory could not be used for other purposes. Furthermore, there is a need for a mechanism to load this table on system start. Using the main memory to store the function length table, the latency for a function length table access could be rather high and accesses could interfere with any other memory requests. The advantage of such a



(a) Eviction on granularity of functions

(b) Fine-grained eviction with non-continuously stored function

(c) Fine-grained eviction with function relocation

Figure 3.4: Different function eviction strategies

table is that the application code does not need to be changed in contrast to instrumenting the functions.

For instrumenting the functions, by the compiler or a post-link tool, a special instruction that contains the function length has to be added at the beginning of every function. By inspecting the first instruction of a function when loading it, the D–ISP *content management* can determine how many further bytes are needed to load. The third option is to obtain the function end while copying it into the scratchpad, e.g. by finding a return instruction or a specific bit pattern. This might be difficult, if a function has multiple return points or the instructions have different alignments.

After the *content management* is finished with creation of the function mapping it can load the function into the scratchpad memory.

Function Load and Eviction

If a function is not found in the mapping table (so called D–ISP miss) the code of the function has to be loaded before the processor can execute it. To load the function the *content management* requests the needed blocks from the memory arbiter that uses its connection to the higher memory level. The received blocks are copied by the content manager into the scratchpad. After all blocks are copied to the scratchpad the context register is written and the *fetch control* triggered to start function execution.

When loading a function into the scratchpad, the *content management* has to check if the free available scratchpad size is larger than the size of the function to load. Otherwise the *content management* has to evict one or multiple functions. Since functions can only be maintained either on the whole or not, the overwriting of only one memory block of a function leads to its eviction from the scratchpad. The drawback of this restriction is that possibly unused memory is invalidated on function load, causing more memory blocks to be reloaded later on function activation. This is depicted in Figure 3.4(a): The function a is evicted due to the load of function b. On the reactivation of a, the whole function is reloaded, whereas only a part of its memory was overwritten by b.

The memory utilisation could be increased, if the *content management* allows to evict parts of a function. But this would result in a complex *content management*, which either has to take care of non-continuously stored functions or it has to relocate functions.

Consider the Figure 3.4(b) in which the function b will overwrite a part of function a. On later reactivation of a the *content management* uses the rest of a that is still present in the memory. So only the missing part of a is to be loaded into the memory. The problem of this approach is that the function is not stored in a sequential continuous order. Hence, the complexity of the *content management* is increased, because on function lookup it has to locate the different

scattered parts of the function. In addition to that, the *content management* has to be aware of functions that are only partly contained in the scratchpad and it has to identify which parts of the functions are missing. This may lead to a fragmentation of functions during runtime. Beside the *content management* the *fetch control* has also to deal with fragmented functions, which would demand a more complex address translation. Increasing the complexity of the *fetch control* might cause additional delay of the fetch process and so increasing the fetch latency.

A better solution would be to relocate and reassemble partly contained functions on their reactivation. By this the *content management* enforces a continuous storage of the functions. Before function reactivation the *content management* finds parts of the function that are still in the scratchpad and selects a valid memory region for the function. Depending on the replacement policy the placement of the reactivated function is not trivial, because the *content management* should not disrupt the eviction order. So a domino effect could be triggered when cascading relocations become necessary when trying to rearrange the scattered parts of different functions. Figure 3.4(c) shows the relocation and reassembly of function a without a cascading function relocation. First the resident function part is relocated, then the missing part of the function loaded. Thus a sequential and continuous function addressing is enforced and the *fetch control* does not need to be adjusted. Nevertheless, the *content management* has to know which function is not completely stored in the scratchpad memory, which part is missing, and of course it has to perform the function relocation. Since the D–ISP controller is intended to be implemented in hardware, this would render the implementation very costly, especially the relocation and the affiliated search for a appropriate memory location. Furthermore, the time needed by function relocation could be worse than simply reloading the complete function, if cascading relocations occur. This is undesirable according to the run-time and from the timing analysis point of view.

The content analysis of the scratchpad, which is needed for the WCET estimation, is another concern when allowing functions to be partly maintained by the D–ISP *content management*. Then the content analysis cannot use functions as logical units, instead it has to consider each memory block of the function separately. This increases the complexity of the analysis and will impact its accuracy, run-time, and required memory effort. The possibility of relocating functions introduces further complexity to the content analysis, especially if cascading relocations might occur. So to keep WCET analysis as simple as possible without loosing precision and also limit the hardware effort of the D–ISP controller the *content management* is restricted to maintain functions only on the whole. Thus a function is defined as evicted, if only a part of it is overwritten. This is not optimal regarding scratchpad memory utilisation, but due to analysis and hardware complexity this is the option of choice. To evict a function the D–ISP *content management* simply needs to invalidate the according entry in the lookup and mapping table. Then on reactivation of the evicted function it will not be found on function lookup and is loaded again by the *content management*. Which function will be evicted on memory shortage is dependent on the replacement policy that is used. Therefore, in the next section different replacement policies for the D–ISP are introduced and their behaviour will be described.

3.2.4 Replacement Policies

The choice of a replacement policy is important for the analysability and the worst-case performance of the D–ISP. In the following the two common replacement policies FIFO and LRU are described for the D–ISP *content management*. Because the D–ISP works on the granularity of functions, it seems likely to employ a stack-based replacement policy which is also introduced in the following. The analysis of the replacement policies for the D–ISP will be discussed in detail in Section 4.3.

FIFO

The FIFO (First In First Out) replacement policy evicts functions in the same order as they were loaded. This is of advantage when the recently loaded functions are used again in the future.

Example. Consider a D–ISP that contains three functions: f, k and l. The scratchpad content is presented as the content of [ ], in which the functions are ordered from new to old, with the oldest function on the right. The size of the scratchpad is 50 and the employed replacement policy is $FIFO$ indicated by the indices on the closing bracket: $[\,]_{50}^{FIFO}$. So the D–ISP content is represented as follows: $[f_{10}, k_{15}, l_{30}]_{50}^{FIFO}$. The right lower index of the function name shows the function size, such that the function f with a size (size(f)) of 10 is denoted as f_{10}.

Assuming this scratchpad content the D–ISP controller detects a miss when activating the function g_{10} and it has to be loaded into the D–ISP content. When adding the function g_{10} to the D–ISP the oldest function (rightmost) has to be evicted, because the size of the functions in the scratchpad including g, exceeds the scratchpad size. See this example and the handling of two other D–ISP misses and one hit below:

$$\begin{aligned}
\text{miss:} \quad & [f_{10}, k_{15}, l_{30}]_{50}^{FIFO} && \xrightarrow{g_{10}} [g_{10}, f_{10}, k_{15}]_{50}^{FIFO} \\
\text{hit:} \quad & [g_{10}, f_{10}, k_{15}]_{50}^{FIFO} && \xrightarrow{k_{15}} [g_{10}, f_{10}, k_{15}]_{50}^{FIFO} \\
\text{miss:} \quad & [g_{10}, f_{10}, k_{15}]_{50}^{FIFO} && \xrightarrow{h_{35}} [h_{35}, g_{10}]_{50}^{FIFO} \\
\text{miss:} \quad & [h_{35}, g_{10}]_{50}^{FIFO} && \xrightarrow{n_{05}} [n_{05}, h_{35}, g_{10}]_{50}^{FIFO}
\end{aligned}$$

♦

The drawback of the FIFO policy is that it does not take into account, if a function is often used or just once. So the activation of function k_{15} in the second line of the example does not prevent it from become evicted on the next D–ISP miss.

LRU

The LRU (Least Recently Used) replacement policy takes the access history of functions maintained by the D–ISP into account. Therefore, on function access the policy sets the age of the activated function to minimum and increases the age of all other functions. On memory shortage functions are evicted with decreasing age. This is in contrast to FIFO in which the order of eviction equals the order of load.

Example. Consider the following example. The representation of the D–ISP content is similar to the example for FIFO. In contrast to FIFO the age of the function is changed on a D–ISP hit. Then age of the activated function is set to minimum. This is illustrated by moving the function to the leftmost position in the representation of the D–ISP content. So the access of function k in line two prevents it from being evicted on miss of function h in the next line:

$$\begin{aligned}
\text{miss:} \quad & [f_{10}, k_{15}, l_{30}]_{50}^{LRU} && \xrightarrow{g_{10}} [g_{10}, f_{10}, k_{15}]_{50}^{LRU} \\
\text{hit:} \quad & [g_{10}, f_{10}, k_{15}]_{50}^{LRU} && \xrightarrow{k_{15}} [k_{15}, g_{10}, f_{10}]_{50}^{LRU} \\
\text{miss:} \quad & [k_{15}, g_{10}, f_{10}]_{50}^{LRU} && \xrightarrow{h_{35}} [h_{35}, k_{15}]_{50}^{LRU} \\
\text{miss:} \quad & [h_{35}, k_{15}]_{50}^{LRU} && \xrightarrow{n_{05}} [n_{05}, h_{35}]_{50}^{LRU}
\end{aligned}$$

♦

Stack-Based

Because the D–ISP works on the granularity of functions, it seems natural to employ a stack-based replacement policy. The basic idea of the policy is that a caller function should be still contained in the scratchpad on return of its callee. Therefore, the stack-based policy tries to lock all functions in the D–ISP memory that are on the active branch of the function call graph. By maintaining the currently active branch in the scratchpad memory, the D–ISP holds only the functions which are guaranteed to be accessed again on return. This is in contrast to the FIFO policy that keeps all functions even if they are executed only once. The stack-based replacement policy has the following behaviour: The scratchpad memory can be seen as a list in which the currently active function is marked. If a function is called, it is supposed to be in the right of the active function. On function return the caller function will be located in the left. Thus, the decision which function is evicted depends on the currently active function and the mode of function activation: on call/return the function to the right/left will be evicted.

Example. Assume an empty memory state. A function call to function a is denoted with call(a_{10}) and a return back to the function a with ret(a_{10}). The currently active function is denoted as $\dot{a}_{10}$. When calling a function that is not located in the D–ISP, the activated function will be added to the right-hand side of the caller function:

$$\begin{aligned}
\text{miss:} \quad & [\,]_{50}^{STACK} && \xrightarrow{\text{call}(a_{10})} [\dot{a}_{10}]_{50}^{STACK} \\
\text{miss:} \quad & [\dot{a}_{10}]_{50}^{STACK} && \xrightarrow{\text{call}(b_{10})} [a_{10}, \dot{b}_{10}]_{50}^{STACK} \\
\text{miss:} \quad & [a_{10}, \dot{b}_{10}]_{50}^{STACK} && \xrightarrow{\text{call}(c_{10})} [a_{10}, b_{10}, \dot{c}_{10}]_{50}^{STACK}
\end{aligned}$$

If a function is already contained in the D–ISP, it is simply activated:

$$\begin{aligned}
\text{hit:} \quad & [a_{10}, b_{10}, \dot{c}_{10}]_{50}^{STACK} && \xrightarrow{\text{ret}(b_{10})} [a_{10}, \dot{b}_{10}, c_{10}]_{50}^{STACK} \\
\text{hit:} \quad & [a_{10}, \dot{b}_{10}, c_{10}]_{50}^{STACK} && \xrightarrow{\text{call}(c_{10})} [a_{10}, b_{10}, \dot{c}_{10}]_{50}^{STACK}
\end{aligned}$$

A function f gets evicted, if a non-maintained function is activated by call and f is on the right-hand side of the active function.

$$\begin{aligned}
\text{hit:} \quad & [a_{10}, b_{10}, \dot{c}_{10}]_{50}^{STACK} && \xrightarrow{\text{ret}(b_{10})} [a_{10}, \dot{b}_{10}, c_{10}]_{50}^{STACK} \\
\text{miss:} \quad & [a_{10}, \dot{b}_{10}, c_{10}]_{50}^{STACK} && \xrightarrow{\text{call}(d_{30})} [a_{10}, b_{10}, \dot{d}_{30}]_{50}^{STACK}
\end{aligned}$$

If the unused scratchpad size exceeds on call, then the function with the largest distance to the current stack position, which is the leftmost function in the list, is evicted.

$$\text{miss:} \quad [a_{10}, b_{10}, \dot{d}_{30}]_{50}^{STACK} \xrightarrow{\text{call}(e_{10})} [b_{10}, d_{30}, \dot{e}_{10}]_{50}^{STACK}$$

On return the caller function has to be on the left-hand side of the function that is terminated. The function reactivation on return will always be a hit, except if the function was evicted before, which as described above can happen only by a call. On a miss during return the function with the largest stack distance to the caller, which is the rightmost function in the list, will be deleted.

$$\begin{aligned}
\text{hit:} \quad & [b_{10}, d_{30}, \dot{e}_{10}]_{50}^{STACK} && \xrightarrow{\text{ret}(d_{30})} [b_{10}, \dot{d}_{30}, e_{10}]_{50}^{STACK} \\
\text{hit:} \quad & [b_{10}, \dot{d}_{30}, e_{10}]_{50}^{STACK} && \xrightarrow{\text{ret}(b_{10})} [\dot{b}_{10}, d_{30}, e_{10}]_{50}^{STACK} \\
\text{miss:} \quad & [\dot{b}_{10}, d_{30}, e_{10}]_{50}^{STACK} && \xrightarrow{\text{ret}(a_{10})} [\dot{a}_{10}, b_{10}, d_{30}]_{50}^{STACK}
\end{aligned}$$

♦

The content management for the stack-based policy is much like for the FIFO replacement policy, with the difference that on both ends of the list evictions can take place (right on call, left on return). Thereby, a function f does not get evicted until the sum of the sizes of the functions on the active branch of f's call graph is larger than the scratchpad size.

Preußer et al. [2007] propose a stack-based replacement policy for the method cache of Schoeberl [2004]. The implementation of the replacement policy uses a doubly linked list of pointers alongside the current branch of the call graph. In contrast to the replacement policy of Preußer et al., the proposed stack-based replacement policy for the D–ISP uses a lookup table that allows to activate every maintained function and not only the ones that are directly reachable by the pointers. This allows to better utilise the memory space, e.g. when a callee function is called by different callers.

3.3 Implementation

In this section the implementation details of the D–ISP architecture proposed in Section 3.2 are described. For architectural evaluation the D–ISP is implemented in a cycle-accurate processor model using SystemC [IEEE Computer Society, 2011]. The simulation model is designed to be easily converted into synthesiseable VHDL to further deploy the D–ISP to an FPGA. Though this conversion is to be done manually, it is of advantage if the timing, signal behaviour, and code itself of the SystemC model can be used as reference for the VHDL generation. Hence, the SystemC model is developed to be as close to an implementation of VHDL in an FPGA as possible.

Before starting with the description of the D–ISP implementation in detail, the architectural requirements and the host processor integration of the D–ISP is discussed.

3.3.1 Implementation Requirements

The D–ISP has some requirements on the host system architecture to allow its integration. The requirements can be split into processor requirements, containing the presence or absence of specific architectural features of the host processor, chip-level requirements, which define especially memory hierarchies, and system requirements, that cover all requirements of the D–ISP beyond the chip.

Processor Requirements

The D–ISP is designed as a first level instruction memory that is located close to the processor pipeline. There are some general requirements that hinder the integration of the D–ISP, if they cannot be meet by the processor. The basic requirement is that the D–ISP prohibits any speculation that leaves the context of a function. This is because the D–ISP maintains its content on the granularity of functions and by definition only one function is allowed to be active.

An example for this is a **branch prediction** that prefetches instructions from a function that will be called on a speculative path. Such branch prediction beyond the scope of a function, would need to identify call and return instructions. Then it has to calculate the target address or it would require a buffer that holds for call instructions the right function addresses and for return instructions the correct return address. The benefit of such branch prediction is not very promising, since the instruction fetch is not timing critical on function call or return. The critical part of function calls and returns is usually the saving and the restoring of the context. Because an implementation of such branch prediction would be very complex and the benefit is doubtful, a branch prediction beyond the scope of a function is not likely to be used in any processor. So

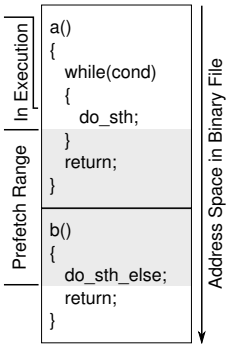


Figure 3.5: Prefetching beyond function borders

if the context of a function is not left by the branch prediction, the D–ISP can be integrated without any further restrictions to the branch prediction.

Like the branch prediction also **instruction prefetching** [Hennessy and Patterson, 2006] is allowed for the D–ISP, if the context of a function is not left. To handle prefetching that may leave the context of an active function when the D–ISP is used, there are three possibilities: (1) Bypass the D–ISP for prefetch requests that leave the function context. This solution is not desirable, because it will disrupt the two-phased execution scheme of the D–ISP and thus increase the complexity of a precise WCET analysis. (2) Allow the D–ISP to trigger the *content management* and activate the function to which the prefetch request is directed. This is complicated because, the *content management* has to detect the function address, perform a lookup, and possibly load the function into the D–ISP. Then it is possible that the currently active function may be evicted, causing a potential reload of the active function.

Example. Assume the functions a and b, which both do not fit the scratchpad together. Further assume that during the execution of the `while`-loop of function a the instruction prefetching is allowed to fetch instructions from b, as depicted in Figure 3.5. This will cause the eviction of a, if the D–ISP is allowed to load b on prefetch requests. Then on consecutive execution of the `while`-loop the D–ISP does not contain the currently active function a and it has to be reloaded. So additional loading effort, which also depends on the iteration count of the `while`-loop, is needed by the instruction prefetching. ♦

Anyhow, if allowing the D–ISP to load non-activated functions, the two-phased execution scheme cannot be enforced, since it is possible that an data access collides with a D–ISP load access triggered by prefetching. Also the complexity of the content analysis for the D–ISP is escalated, because then the main restriction that the currently active function is always contained in the scratchpad is void. Furthermore, content changes are triggered by speculation, which has also a negative impact the analysability of the system. (3) The D–ISP ignores any fetch requests that leave the function and answers it with invalid data. Usually these speculatively fetch blocks are not executed, except if the function is left and the particular function that was prefetched is executed. To ensure that no invalid fetch blocks are executed in this case, the D–ISP requires an instruction buffer flush on call and return. Then any prefetching scheme can be applied by the processor. This option preserves the timing and content behaviour of the D–ISP and adds only a minor additional fetch effort due to the instruction buffer flush on function activation, because it is not likely that the code of the activated function was prefetched before. Hence, this solution is the option of choice to support both: instruction prefetching and the D–ISP.

For **interrupts** the D–ISP has to be disabled while the execution of the interrupt service routine (ISR), because otherwise at any point in program execution an interrupt could disrupt the content of the D–ISP memory and a WCET analysis taking the D–ISP content into account would be not applicable. Therefore, the best solution is to disable the D–ISP on interrupts and store the ISRs into a separated memory partition that is set up on the initialisation of the system. The necessary instructions for the ISR could be fetched from the next memory level or an additional scratchpad. Using the next memory level is not recommended, because the interrupt handling will be slowed down and the timing analysis has to assume the collision of instruction and data access for the execution of the ISR, causing a higher worst-case memory latency of every fetch and load/store. Therefore, the usage of an additional scratchpad containing the ISRs is preferable.

If the processor features **multithreading**, either the D–ISP can be granted to one thread only or the D–ISP needs the ID of the thread that invokes a call or return, in case the system supports multiple hard real-time threads. To preserve timing analysability both approaches require that the D–ISP is used only by hard real-time threads that are isolated of other lower priority threads. If the application consists of multiple hard real-time threads that use the D–ISP, it is furthermore required that they are fully isolated, e.g. as proposed by Mische et al. [2008]. It is recommended that the hard real-time threads use separated D–ISP memory partitions, such that the content analysis is similar to the analysis of a D–ISP in a single threaded processor. If a shared D–ISP memory is used, threads can interfere with the scratchpad content used by other threads, e.g. by evicting function of other threads. This would render the D–ISP content analysis very complex or utterly pessimistic.

Any other features employed in embedded high performance processors can be used together with the D–ISP. Especially for instruction level parallelism (like superscalar pipelines with in-order or out-of-order execution) or multicore processor architectures. There are no restrictions of the D–ISP with respect to the constraint that no speculation will leave the active function.

Chip-Level Requirements

The D–ISP has to be added onto the fetch path of the processor. By design it is intended to replace a standard instruction cache. Additionally some extra signals of the processor pipeline are needed to correctly control the D–ISP *content management*. It is also possible to add the D–ISP beside another custom or standard instruction memory to enable and use the D–ISP only for specific parts of the application. This option is depicted in Figure 3.6. Notice that an additional fetch multiplexer is required that routes the fetch results from the correct memory to the pipeline. It is intended that the fetch multiplexer is controlled by the D–ISP. Fetches are directed to the D–ISP when it is activated, otherwise the other memory structure is used for handling the instruction fetches.

If the processor uses a shared off-chip memory connection for instruction and data memory controlled by a memory arbiter, the memory arbiter has to be enhanced to support another master to serve the D–ISP. In case of separated memory connections a memory arbiter is only needed, if the D–ISP shall be used with other instruction memories, like an instruction cache. Otherwise the D–ISP may directly access the higher memory level.

So beside modifications of the on-chip memory arbiter and an additional fetch multiplexer no further changes are necessary to the use of the D–ISP in conjunction with any kind of on-chip data and instruction memories.

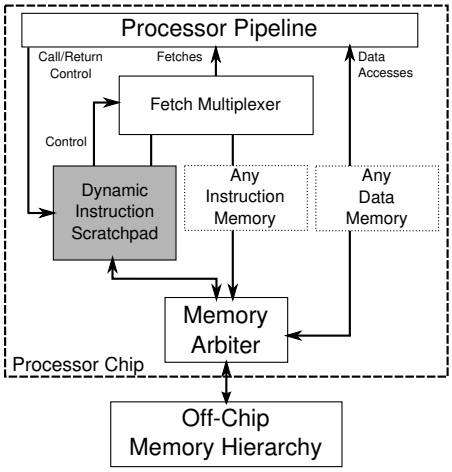


Figure 3.6: D–ISP chip integration overview

System Architectural Requirements

The task scheduling of the processor has also to be considered when using the D–ISP as first level instruction memory. Non-preemptive task scheduling schemes are supported by the D–ISP by design. But for a preemptive scheduling, that change the context by switching from one task to another during its execution, the D–ISP has to be considered. On a task switch the execution context changes and the D–ISP controller has to be aware of the context of the task to which the focus is moved. So as the processor context, including the register set, has to be maintained by the scheduler also the context of the D–ISP needs to be managed. Namely the *context register* that defines the active function and the *context stack* memory, which holds the stack of the activated functions, is to be saved and restored on task switch. Additionally on task activation the D–ISP has to check its content for the presence of the activated function.

For a WCET analysis the scheduling policy should be analysable and the content of the scratchpad memory has to be considered on task switch. If the WCET analysis cannot determine the task switch points in the execution of the tasks, it has to consider that the D–ISP content is in its worst-case state, which depends on the applied replacement policy. Then the D–ISP has to implement a *content management* that's behaviour is independent of the (obsolete) content of the deactivated task. Otherwise a method to flush the D–ISP content is required.

Summary of Restrictions for the D–ISP

Below the main restrictions that are of an D–ISP integration into a given system are shown:

- Control flow speculation beyond function borders is prohibit.
- Instruction buffer flush on call/return is required, if the processor implements prefetching.
- Shared D–ISP memory partitions between multiple threads or tasks are not recommended.
- An additional connection to a on-chip memory arbiter is required, if a shared on-chip memory connection is used.
- If a preemptive task management is used, the D–ISP and the scheduler require additional support for context save and restore.

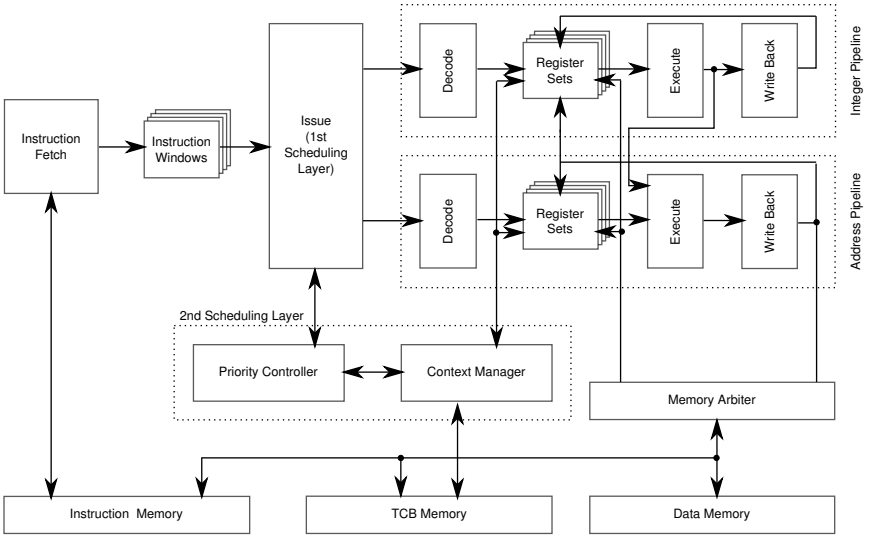


Figure 3.7: Block diagram of the CarCore processor (according to [Mische, 2011])

3.3.2 Host Processor Integration

The D–ISP is implemented in the single-core CarCore processor and in its multicore version the MERASA processor. In this section these processors are briefly described. Also the modifications that are needed to integrate the D–ISP into these architectures will be discussed.

The CarCore Processor

The CarCore processor was developed in the CAR-SoC project as an simultaneous multithreaded hard real-time capable processor. It is inspired by the Infineon TriCore processor [TriCore, 2007a]. The TriCore is a high-performance embedded processor for automotive applications, it combines the characteristics of a RISC, CISC, and a DSP processor. Since the instruction set of the TriCore supports more than 700 instructions, the CarCore implements only the subset of the instructions that is generated by the Hightec GCC compiler [HighTec] and Tasking compiler [Altium, 2009]. Anyhow, the number of instructions supported by the CarCore is with 433 rather high.

The CarCore features two four-staged pipelines, one for address calculations and one for integer arithmetics. The additional loop pipeline that the TriCore processor has is not implemented in the CarCore. The Figure 3.7 gives a deeper insight into the CarCore architecture. In contrast to the TriCore processor the CarCore processor is simultaneous multithreaded (SMT). The differences to the TriCore and implementation of the CarCore architecture are described in detail in [Mische et al., 2010a; Mische, 2011].

The issue stage of the CarCore processor selects which thread is fed into which pipeline. It uses a fixed priority scheme for the scheduling of the different threads. One thread is able to be issued into both pipelines within one cycle, if an integer pipeline instruction is followed by an address pipeline instruction. The priority list of the issue stage is controlled by the second scheduling layer, that allows to apply different real-time scheduling schemes as Mische et al. [2008; 2009] proposes. The CarCore base architecture has separated instruction and data memories.

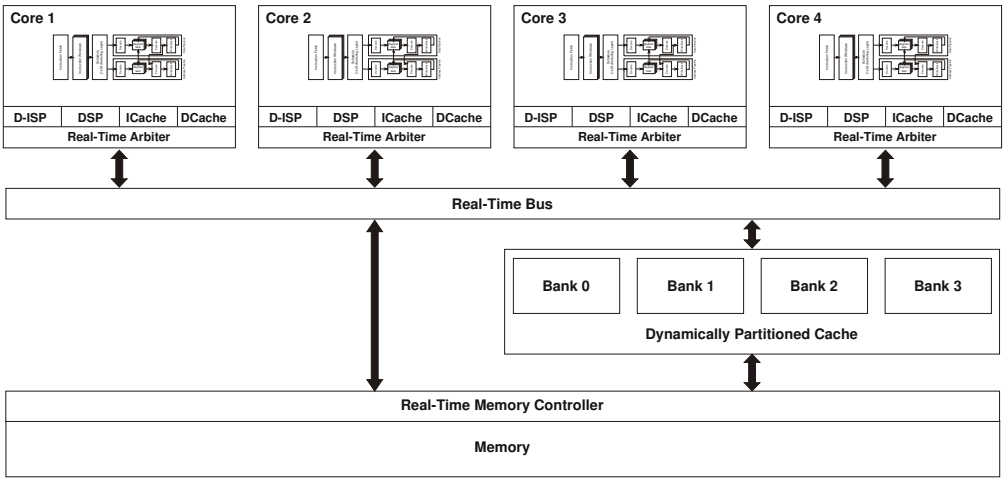


Figure 3.8: MERASA processor overview (according to [MERASA, 2009a])

Also an additional thread control block (TCB) memory is used. It holds information about the threads and is used as an interface to the second scheduling level. The CarCore does not support instruction prefetching or branch prediction to keep the timing model of the processor simple.

The MERASA Processor

The MERASA processor is a multicore processor designed for predictable execution of hard real-time applications. It is fully analysable, which is proved by static [Ballabriga et al., 2011] and measurement-based [RapiTime] WCET tools. The MERASA processor also allows the analysable execution of parallel applications described by Wolf et al. [2010; 2011] and Gerdes et al. [2011].

The MERASA core architecture is similar to the CarCore processor except for the scheduling and the memory hierarchy. The MERASA core features one hard real-time thread and multiple soft/non real-time threads, whereas the CarCore allows multiple hard real-time threads by Dominant Time Slicing [Mische et al., 2010b]. In contrast to the CarCore processor the MERASA core features local memories to buffer data and instructions. For hard real-time threads the D–ISP and a data scratchpad (DSP) is used. Since usually a large fraction of the data accesses are stack accesses [Deverge and Puaut, 2007b], the application's stack is maintained by the DSP. The impact of the DSP is discussed in more detail in [MERASA, 2009b; Paolieri et al., 2012]. Non hard real-time threads use first level caches for data and instructions. The core architecture and the overall MERASA multicore chip architecture is shown in Figure 3.8. The architecture of the MERASA processor is described in detail in [Ungerer et al., 2010; Paolieri et al., 2012].

Beyond the core level the MERASA processor features a real-time capable bus to connect the cores to the shared next level cache or the off-chip memory. The bus policy is TDMA-like and thus enforces a known upper bound for each memory request [Paolieri et al., 2009b]. To isolate the memory accesses of different cores in the next level cache the MERASA processor uses cache partitioning. The isolation ensured by partitioning is crucial for a precise timing analysis, since it excludes interferences among different threads that cannot be taken into account by the cache content analysis without a high amount of pessimism.

A timing bound for off-chip memory accesses is also of importance for an analysable architecture especially, if DRAM is used as off-chip memory, which is discussed by Atanassov and

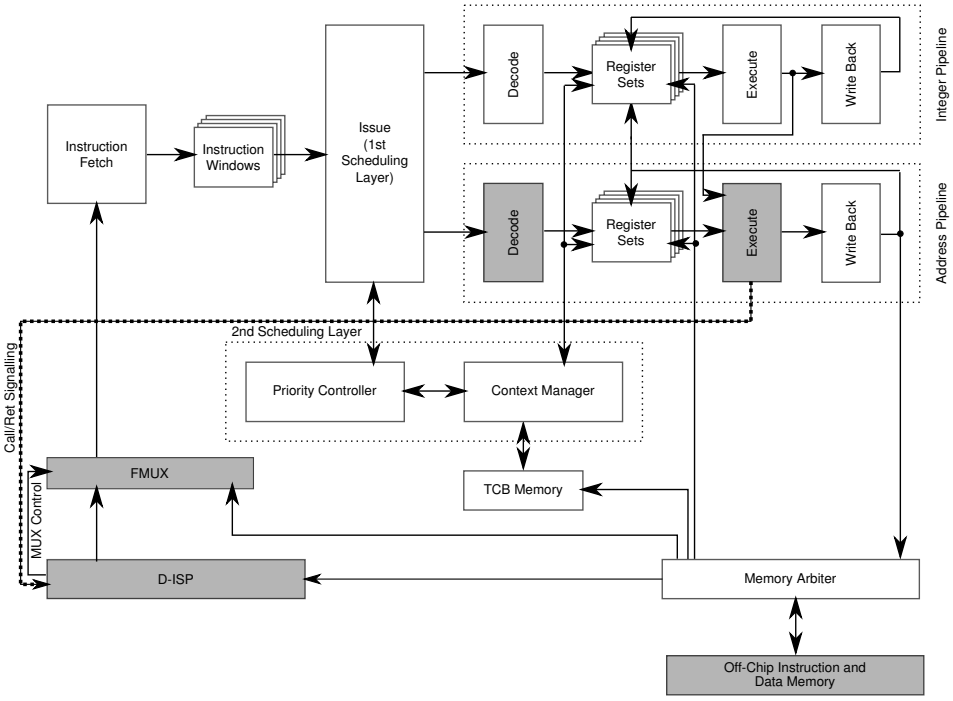


Figure 3.9: Block diagram of the CarCore processor with integrated D–ISP (changes to original architecture are highlighted)

Puschner [2001] and Akesson et al. [2007]. To allow an off-chip memory access with a known timing a custom memory controller timing model for DRAMs is employed in the MERASA architecture. The designed memory controller enforces a safe upper bound for each memory access [Paolieri et al., 2009a].

Changes to the CarCore Base Architecture

The basic CarCore architecture does not distinguish different memory levels [Mische et al., 2010a]. To integrate the D–ISP and show its benefits for a shared off-chip memory connection an off-chip memory containing instructions and data is added. This off-chip memory can only be accessed by the memory arbiter, at which interferences between data and instruction memory accesses can occur. In case of a conflict the data memory access is preferred to keep the pipelines running, whereas the instruction fetch request is delayed. The D–ISP is located between the fetch stage of the processor pipeline and the memory arbiter that connects to the off-chip memory. The changes to the CarCore base architecture made for the D–ISP are shown in Figure 3.9.

The D–ISP is connected to the memory arbiter to obtain the code of the functions on their activation. Therefore, the memory arbiter needs either an additional port for the D–ISP. Otherwise the port that is used by the pipeline for fetches can be shared, because the fetches from the pipeline will be disabled by the D–ISP as long as it is active.

To allow bypassing the D–ISP on instruction fetch a fetch multiplexer ($FMUX$) is used. It is controlled by D–ISP controller and directs all fetch requests to the D–ISP and the fetch results of the D–ISP back to the processor pipeline. The FMUX is used to integrate the D–ISP in a

transparent way: If the D–ISP is inactive, all fetch requests are routed to the memory arbiter and the processor behaves like a system without D–ISP memory.

The FMUX is implemented using an address range for which all requests are directed to the D–ISP. The range spans all possible instruction addresses of the active function. All requests that does not fit the address range are directed to the memory arbiter. The address range is set by the D–ISP *content management* on function activation. Thus, it is ensured that fetch requests which lead to the currently active function will always lead to the D–ISP. Any other fetch requests are handled by the memory arbiter. Notice that this is for compatibility reasons, if e.g. an interrupt service routine has to be executed or the application intentionally leaves a function address range by a jump, which is not recommended (see Section 5.3 for application requirements). To correctly support prefetching, without the disruption of the two-phased execution scheme the function address range is extended to route all fetch requests to the D–ISP.

Because the CarCore processor supports SMT, the FMUX is configured per thread. So it can direct fetch requests to the D–ISP or to the memory arbiter by distinguishing the thread ID of the fetch request. It is also possible that multiple D–ISPs are implemented to physically separate the contents of different threads and the FMUX routes for each thread the fetch request to the corresponding D–ISP. For simplicity it is assumed in the rest of this work that only one hard real-time thread is present using the D–ISP. All other threads are intended to bypass the D–ISP.

To be informed of function activation the D–ISP controller has to detect the invocation of calls and returns, distinguish them, and be aware of the addresses of the functions that are called. To provide these information the decode stage has to be extended to drive signals for call and return instructions. Then these signals are routed to the execute stage that provides them as output to the D–ISP. Notice that only the decode and execute stage of the address pipeline is affected, because the call and return are address instructions that are handled by the address pipeline. To access this information the D–ISP snoops these signals for call and return detection and additional signals from the execute stage like the jump target address. The Figure 3.9 highlights the path of the necessary signals from the CarCore processor pipeline to the D–ISP. These signals are limited to:

- **Call/Return notifications**: Additional boolean signals for call and return set by the decode stage on instruction decode.
- **Call target address**: Result of address calculation for calls provided by execute stage.
- **Branch notification**: Signal for an active branch that is available by default in the execute stage and is needed to flush instruction window on jump.
- **Thread ID**[4]: Signal which is necessary to distinguish different threads and which is available by default in the execute stage.

If the decode stage should not be modified, it is also possible that the D–ISP decodes the instructions processed by the pipeline and detects calls and returns on its own. For complexity reasons of the D–ISP controller this option is not considered.

The CarCore base architecture with integrated D–ISP builds the basic of the MERASA core architecture. Using D–ISP in the MERASA multicore processor is intended to allow a safe WCET estimation and also ease the increased pressure on the memory system introduced by a multicore architecture. Within this work the focus is on the CarCore processor, refer for a discussion and evaluation of the D–ISP for the MERASA processor to [MERASA, 2009b; Ungerer et al., 2010; Paolieri et al., 2012].

[4]By default there is only one hard real-time thread that uses the D–ISP.

3.3.3 Fetch Control

The *fetch control* of the D–ISP accesses the *context register* that is written by the *content management* to map the fetch address requested by the processor to the position where the corresponding function is stored in the scratchpad. Therefore, the context register holds the function address of the active function in native address space (CR_N) and in scratchpad address space (CR_D). To detect if a fetch request is out of the range of the active function, it also contains the function size (CR_S). Fetch requests, which leave the function's address range, can occur, if prefetching is used or the function is left by a jump or any exception like an interrupt. Furthermore, the validity (CR_{valid}) of the context register is stored. The validity is a bit that is set by the *content management* after activating a function. While the *content management* is active the content of the context register is invalid.

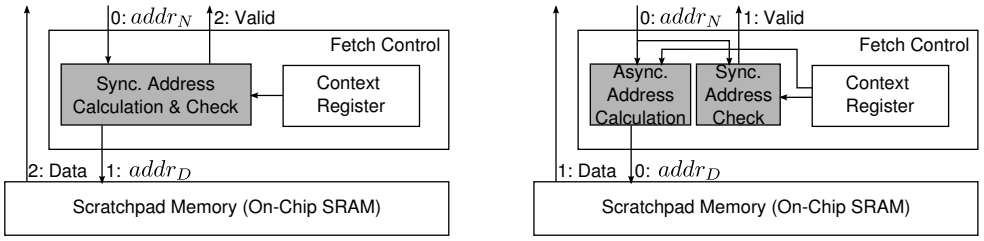
Depending on the address of the fetch request the *fetch control* either confirms or rejects the fetch requests. The confirmation is shown in Equation (3.2). Notice that if a requested fetch leaves the function, but is still in the prefetch range (PF_R, in byte) it is also treated valid, although the corresponding fetch result will not contain the correct instructions. Anyhow, this is not disadvantageous, because the prefetched instructions will not be executed. This would require to leave the context of the currently active function, which causes a flush of the processor's instruction buffer and deletes the prefetched instructions. The calculation of the address itself was introduced with Equation (3.1) in Section 3.2.2. Because the D–ISP has a limited size (size($D\text{–}ISP$)) and functions can be mapped anywhere in the scratchpad memory, it is possible that a function wraps around the scratchpad address space. Therefore, the *fetch control* uses cyclic addressing that maps addresses leaving the D–ISP's address space to its beginning. To efficiently implement this, the D–ISP size needs to be a power of two. Then the D–ISP addresses can be restricted to use only $\log_2(\text{size}(D\text{–}ISP))$ bits and the cyclic addressing is achieved implicitly. Equation (3.3) shows the address calculation performed by the *fetch control*. Notice that the validity check of the fetched address is done only by Equation (3.2).

$$fetch\_valid(x) = \begin{cases} CR_{valid} & \text{if } CR_N \leq addr_N(x) \leq CR_N + CR_S + PF_R \\ false & \text{otherwise} \end{cases} \quad (3.2)$$

$$addr_D(x)|_{\log_2(\text{size}(D\text{–}ISP))} = addr_N(x) - CR_N + CR_D \quad (3.3)$$

The *fetch control* is connected to the processor front-end, namely the fetch request and fetch result signals. The *fetch control* needs to route the calculated address to the internal scratchpad memory that contains the functions. Because the scratchpad memory needs at least one cycle to handle the fetch request and the fetch latency is critical for the overall processor performance, the *fetch control* is optimized to provide a continuous instruction stream. This is done by pipelining the *fetch control*: Fetch requests are buffered, such that the processor front-end does not stall on function execution. The number of buffered fetch requests depends on the latency of the scratchpad memory: For each cycle of the on-chip scratchpad memory access time, one fetch request has to be buffered by the *fetch control*. For the CarCore processor one buffered fetch request is sufficient, because an access time of a single cycle is realistic for first level memories in embedded processors.

Furthermore, to speed up the fetch process the *fetch control* can also be implemented asynchronously in hardware. This is possible, because the necessary calculations are very simple (see Equations (3.2) and (3.3)). A structural overview of a synchronous reference implementation is shown in Figure 3.10(a). There the address calculation takes one cycle and thus the result from the scratchpad memory is delivered after two cycles to the processor. Therefore, the corresponding valid signal needs to be delayed too. Due to the pipelining of the fetch control the D–ISP delivers during function execution one fetch result per cycle.



(a) Synchronous *fetch control* (b) Asynchronous *fetch control*

Figure 3.10: Implementation variants of the D–ISP *fetch control* (The numbers at the description of the signals determine the clock cycle at which the correct values are provided, when receiving the fetch address at cycle 0.)

To lower the fetch latency of the D–ISP the address calculation is implemented asynchronous, such that the scratchpad address is propagated to the scratchpad memory within one cycle. Then the memory delivers the fetch result after one cycle. The address checking can be done synchronous, because the valid signal needs to be delivered after one cycle. Notice that independent of the validity of the fetch address, the request is directed to the scratchpad memory. The correctness of the results is guarded by the synchronous address check (see Equation (3.2)). It also invalidates all requests, if the *content management* is active. The asynchronous *fetch control* is depicted in Figure 3.10(b). Using this implementation every fetch is handled as fast as a fetch to any non-dynamically managed scratchpad without address translation. Indeed, the asynchronous *fetch control* reduces the cycle count of a fetch request to minimum, but the process of address calculation implies a small (but not negligible) delay that increases the overall delay of the fetch path. Thus, if the fetch path is the timing critical path of the processor architecture, the maximum clock rate of the processor could be affected. Then it might be worthwhile to use the synchronous implementation instead. A quantification of the impact of the fetch control on the timing is shown in Section 6.1.

3.3.4 Content Management

The main objective of the *content management* is to ensure that functions will be in the scratchpad memory after their activation. The *content management* is triggered by call and return signals, which are set by the processor pipeline on call and return detection. Also the call address is needed by the *content management*. These signals are depicted in Figure 3.11. Also additional control signals to enable or disable the D–ISP are shown, which will be discussed later. The Figure 3.11 also shows the connection of the *content management* to the context register (CR), which holds all data of the currently active function needed by the *fetch control*. This register is written by the *content management* after the *content management* successfully prepared the activated function. Until this is done a valid bit in the context register (CR_{valid}) prevents the *fetch control* from answering fetch requests.

Helper Memories

The D–ISP *content management* needs additional memories to store information of maintained functions. These helper memories are depicted in Figure 3.11. For the helper memories SRAM with access latency of 1 cycle for read and write is used. To allow faster access the helper memory

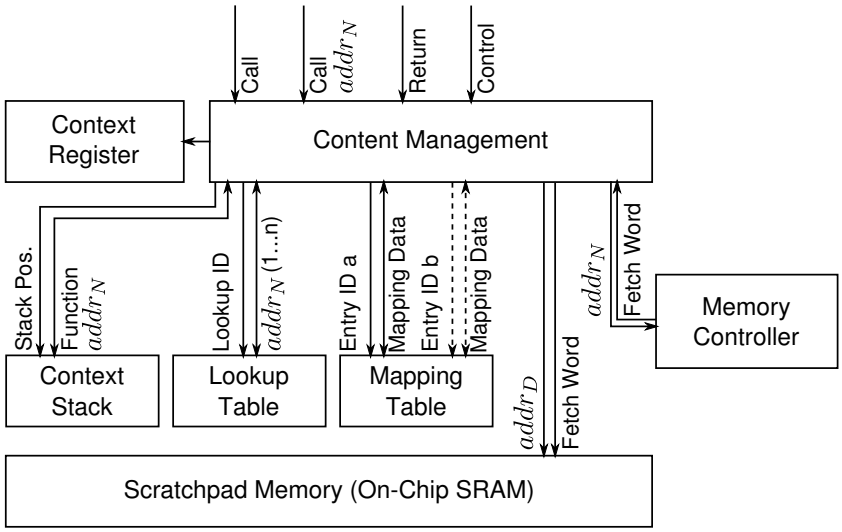


Figure 3.11: Structural view of the D–ISP *content management* containing all its connections

structures could be implemented as registers, but this would escalate the used hardware amount for the D–ISP controller.

The **mapping table** holds the information to find a function in the scratchpad memory. It is also used by the replacement policy to invalidate functions on their eviction. The mapping table contains a certain number of entries (no_{func}) that contain the function address in native ($addr_N$) and scratchpad ($addr_D$) address space and also the function length. The size of the mapping table memory (MTM) is defined by the width of both addresses, the needed memory to store the maximal allowed function length in byte, and the number of entries, as Equation (3.4) shows:

$$\text{size}(MTM) = \left(width_{addr_N} + width_{addr_D} + \left\lceil \log_2(\max(\text{size}(function))) \right\rceil \right) \cdot no_{func} \qquad (3.4)$$

The Figure 3.11 depicts the connection of the *content management* to the mapping table memory with two ports. In principle only one port is necessary, but the second port is used for eviction detection, which is discussed later on.

On call or return the *content management* has to check, if the activated function is already available in the scratchpad. This can be done by comparing all mapping table entries (no_{func}) sequentially. But this will be very slow, because the whole table have to be processed and the fetch of each entry needs one cycle. This results in a large lookup time on function activation. Therefore, an additional **lookup table** is used by the *content management*. The lookup table holds only the addresses of the functions in native address space ($addr_N$). By the ID of an entry in the lookup table, the corresponding mapping table entry is identified. The lookup table memory has a wide output port, allowing that multiple (no_{look}) function addresses can be delivered to the *content management* at once. Then the *content management* compares these addresses to the activated function. On lookup table hit, it selects the corresponding mapping table entry and after one cycle it is able to accesses the mapping information. On miss the next fraction of the lookup table is requested.

The time the function hit detection using the lookup table takes depends on the number of parallel comparisons given by no_{look} and the number of entries in the lookup table, which is defined by no_{func}. Furthermore, depending on the position of the activated function in the

lookup table the hit detection time varies. The upper bound is the time when the whole table is exhaustively checked, which happens on a miss or on a hit for the case that the activated function is at the end of the lookup table. To compare the timing for function hit detection when using the mapping table only ($t_{hit\_detection\_sequential}$) with the usage of the additional lookup table ($t_{hit\_detection\_lookup\_table}$) in the equations below the lower and upper bounds for the hit detection are shown.

$$1 \quad \leq t_{hit\_detection\_sequential} \quad \leq no_{func} \tag{3.5}$$

$$2 \quad \leq t_{hit\_detection\_lookup\_table} \quad \leq \frac{no_{func}}{no_{look}} + 1 \tag{3.6}$$

Notice that an additional cycle is needed by the hit detection using the lookup table, because the mapping table needs one cycle to deliver the data from selected mapping table entry after the function was found in the lookup table.

The speedup of the function hit detection using the lookup table is defined by the number of parallel comparisons (no_{look}), i.e. the number of function addresses that is delivered by the lookup table memory within one cycle. It is shown below in Equation (3.7).

$$speedup_{lookup\_table} = \frac{\max\left(t_{hit\_detection\_sequential}\right)}{\max\left(t_{hit\_detection\_lookup\_table}\right)} = \frac{no_{func}}{\frac{no_{func}}{no_{look}} + 1} \tag{3.7}$$

The optimal lookup time would be reached, if the whole lookup table could be compared at once. In that case the function activation on hit would take only two cycles, but the high number of parallel comparisons requires a tremendous hardware effort for the *context management*. The trade-off between lookup width and used hardware amount of the function hit detection will be discussed in detail by Section 6.1.

The size of the lookup table memory (LTM) is delimited by the width of the addresses in native address space and the number of entries of the mapping table:

$$\text{size}(LTM) = width_{addr_N} \cdot no_{func} \tag{3.8}$$

The last helper memory needed by the *content management* is the so called **context stack**, that is also shown in Figure 3.11. It holds the function addresses in native address space of the in the current call context. On a function call the corresponding address is pushed onto the stack and on return it is obtained. Due to the context stack it is possible for the *content management* to know which function is reactivated on return. The context stack memory always delivers the caller function of the currently active function ($top - 1$ element of the stack), if the *content management* is idle. Then it is possible for the *content management* to identify the function to activate on a return without any delay. So the mapping table hit detection for returns takes the same number of cycles as for calls, for which the address of the activated function is delivered by the pipeline. The size of the context stack memory (CSM) is defined by the maximum stack depth (no_{stack_depth}) supported by the host processor and the width of the addresses in native address space:

$$\text{size}(CSM) = width_{addr_N} \cdot no_{stack\_depth} \tag{3.9}$$

Content Management Finite State Machine

To perform the different objectives like hit detection, function load, and function size determination, the *content management* is implemented as a finite state machine (FSM). The state graph of the *content management* is depicted in Figure 3.12. Because the *content management* is implemented synchronous, each transition from one state to another takes one cycle.

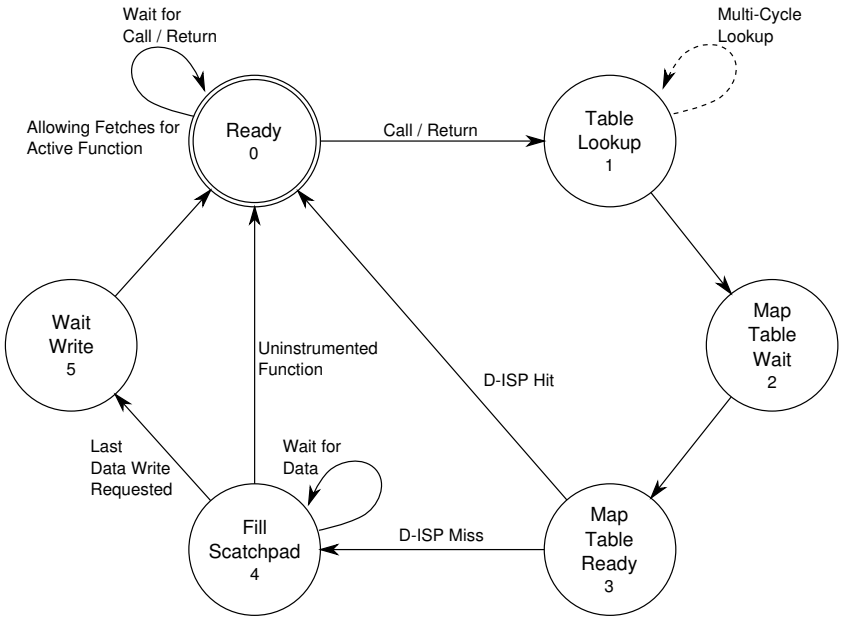


Figure 3.12: D–ISP *content management* state graph

Notice the state graph in Figure 3.12 presents the *content management* that uses the lookup table. If the hit detection would be implemented by an exhaustive search of the mapping table, the state **Table Lookup** would not be needed. Furthermore, an additional self-transition for **Map Table Wait** would be required, in which the entries of the mapping table are sequentially accessed.

The initial state of the *content management* is the state **Ready**. Only in this state the context register is valid and the *fetch control* is allowed to answer fetch requests. In the following the transitions of the graph are described in detail. Thus the behaviour of the *content management* implementation is presented.

- **Ready** → **Ready**: The D–ISP is in function execution mode. So the context register is valid and fetches to the D–ISP are allowed. Also the D–ISP can be configured by the processor.

- **Ready** → **Table Lookup**: A call or return is recognized by the *content management* while function execution. The provided call target address on call respectively the caller address on a return (which is obtained from the context stack) is used as function address. On return the head element of the context stack is deleted, whereas on call the function address is pushed onto the top of the stack. The lookup table is queried for function hit detection.

- **Table Lookup** → **Table Lookup**: If the *content management* uses the multi-cycle lookup and checks only a fraction of all entries in the lookup table per cycle, multiple queries have to be sent to the lookup table. Otherwise this transition is not possible. Moreover, requesting a further part of the lookup table is only required, if the function address was not found in the fraction that is delivered by the lookup table.

- **Table Lookup → Map Table Wait**: If the function address matches an address obtained from the lookup table, the corresponding mapping table entry is requested. If the whole table was exhaustively checked and the activated function was not found in the lookup table, the creation of a new mapping is prepared.
- **Map Table Wait → Map Table Ready**: An additional latency cycle is needed to obtain the content of the selected mapping table entry on a hit. For a miss this cycle is needed to prepare the eviction detection for the replacement policy.
- **Map Table Ready → Ready**: On a D–ISP hit the content of the mapping table is copied to the context register. Furthermore, the context register is set to valid and the *fetch control* is released. Also the FMUX is set to route all fetch requests regarding the activated function to the D–ISP.
- **Map Table Ready → Fill Scratchpad**: If the activated function is not in the scratchpad memory, the mapping and lookup table entry is written and the *content management* starts loading the function by requesting the first fetch block from the memory arbiter.
- **Fill Scratchpad → Fill Scratchpad**: The *content management* stays in this state until the last but one fetch block is received. If a fetch block is delivered by the memory arbiter, it is rerouted to the corresponding scratchpad memory address. Also the next fetch block is requested. The replacement policy ensures that on overwriting of at least one block in the scratchpad memory the corresponding lookup and mapping table entry is cleared.
- **Fill Scratchpad → Wait Write**: The last fetch block was received and is rerouted to the scratchpad memory. An additional cycle for write into the scratchpad memory is needed before the *fetch control* can be released.
- **Wait Write → Ready**: The context register is made valid with the data of the currently active function. Thus, fetching of the active function is enabled. Furthermore, the FMUX is ordered to route all fetch requests for the activated function to the D–ISP.

The latencies introduced by the D–ISP for lookup and function load can be partly hidden by call/return processing of the pipeline, because this is done in the CarCore processor via microcodes sequences and take several cycles [Mische et al., 2010a].

The minimal time for function activation on a D–ISP hit when comparing all functions at once is four cycles. Using the multi-cycle lookup the function activation time on hit is the time to detect the hit using the lookup table, see Equation (3.6), plus two cycles for triggering the lookup (Transition: **Ready → Table Lookup**) and context register write (Transition: **Map Table Ready → Ready**).

$$t_{activation}|_{hit} = t_{hit\_detection\_lookup\_table} + 2 \tag{3.10a}$$

$$t_{activation}|_{hit} \leq \frac{no_{func}}{no_{look}} + 3 \tag{3.10b}$$

On a D–ISP miss the same time to activate the function is needed as for the worst case for a hit plus the time to load the function and one extra cycle as write latency (Transition: **Wait Write → Ready**):

$$t_{activation}|_{miss} = \max\left(t_{hit\_detection\_lookup\_table}\right) + 2 + t_{mem\_access} \cdot \text{size}(function) + 1 \tag{3.11a}$$

$$t_{activation}|_{miss} = \frac{no_{func}}{no_{look}} + t_{mem\_access} \cdot \text{size}(function) + 4 \tag{3.11b}$$

The latency of the off-chip memory (t_{mem_access}) access is fixed and defined by the used off-chip memory and the size of the activated function (size($function$)) is application dependent. Thus, the maximum activation time on miss can be determined a priori by the WCET analysis. Notice that the time for the function activation on miss takes a constant number of cycles. This is due to the hit detection, which needs always the maximum number of cycles on miss, because the whole table has to be processed to verify that the activated function is not present.

Function Size Determination

To obtain the function size on function activation two possibilities were implemented: the detection of the function end on the fly [Metzlaff et al., 2008] and the instrumentation of a length encoding instruction [Metzlaff et al., 2011a].

In the first implementation the D–ISP *content management* detects the function end by snooping the instruction stream on function load. If the end of the function is detected, the function load is terminated and the mapping table entry is updated with the determined function size. To identify the end of each function a special marker, the so called Function Delimiter Bit Pattern (**FDBP**) is used. The FDBP is inserted into the application code e.g. by the instrumentation tool described in Section 5.1. By the fact that this marker is located after the `return` instruction of a function, it is ensured that it will not be not executed by the host processor pipeline. Thus, an enhancement of the instruction set is not necessary. Detecting the FDBP while snooping the obtained fetch blocks on function load requires the FDBP to be unique and must not be part of other instructions. Because the TriCore ISA features instructions with different lengths, the last instruction of a function may have different alignments in the fetch block. Thus it has to be ensured that the FDBP will be always fill at least one fetch block. Then the function end can be easily determined by comparison of the whole block with the FDBP. Using a shorter or not-unique bit pattern or instruction raises ambiguity about function end detection, because the instruction alignments are unknown to the *content management*.

The FDBP has two advantages against scanning for the `return` instruction that can also determine the end of the function. First it allows the programmer respectively the compiler to create multiple return points for one function and second it is easier to determine, because the different alignments of the `return` instruction of a function can be ignored.

To ease the complexity of the D–ISP *content management* and allow the exclusion of functions that e.g. will not fit the scratchpad memory, the function size determination by a special instruction that contains information about the function size is used. The so called Function Length Encoding (**FLE**) instruction is inserted by an instrumentation tool, described in Section 5.1, on the very beginning of the functions. Then when obtaining the first fetch block on function load the *content management* decodes the FLE instruction and requests the owing fetch blocks. For simplicity of the FLE decoding it is possible to force the linker on application build to always align the FLE instruction to the beginning of the first fetch block of each function. This eases the hardware effort of the D–ISP controller, because there is only one possible position of the FLE instruction that have to be taken into account.

The FLE instruction must not to change the processor state, if it is executed by the processor. To guarantee this the *content management* may swap the FLE instruction with a harmless instruction, like a NOP, when writing the first fetch block into the scratchpad memory. Thus no instruction set enhancement for the FLE instruction is necessary. In fact the implemented D–ISP controller does not swap the FLE instruction, because of hardware complexity reasons and the possibility to select an instruction of the TriCore ISA that allows encoding the function length while not changing the processor state on its execution. For details on the selected instruction see Section 5.1.

If the *content management* detects that the function size exceeds the size of the scratchpad it omits loading the function and disables the *fetch control* for this function by setting the FMUX to bypass the D–ISP. Also if the FLE instruction is missing on the beginning of the function that is to be loaded into the scratchpad memory, the *content management* ignores the activated function. This is not recommended, since then the two-phased execution scheme for WCET analysis cannot be applied. But for compatibility of legacy code or if the programmer explicitly decides not to use the D–ISP in certain parts of the application, e.g. that are not timing critical, this behaviour is useful. If using the FDBP to detect the function size, the selection of which functions are put into the D–ISP an which not is not possible.

Replacement Policies

Due to the fact that the functions of an application are not equally sized, an arbitrary replacement policy is complicated to implement in hardware. If parts of the memory have to be moved to implement the replacement policy correctly, additional overhead in hardware complexity and timing has to be taken into account. With this in mind the proposed replacement policies from Section 3.2.4 are revised for there implementability in the *content management*.

The **FIFO** replacement policy can be easily implemented using a cyclic memory addressing model for the scratchpad memory. A single *Write Pointer* is used to determine the position in the scratchpad memory that is overwritten next. By loading a function into the scratchpad this pointer is increased for each fetch block that is written. If the pointer reaches the end of the scratchpad, it is reset to the beginning of the scratchpad memory. Because the write pointer is only moved on writing functions into the scratchpad, it will always overwrite the oldest function in memory, i.e. the replacement is performed in a FIFO manner.

The *content management* has to keep track of the overwriting of functions in the scratchpad memory. By the write pointer, the beginning of a function is overwritten first. Therefore, on overwriting the first block of a function the whole function has to be invalidated, to apply to the restriction of indivisibility of the functions in the scratchpad memory, which is described in Section 3.2.3. Hence, the *content management* has to be aware of the scratchpad address of the oldest entry in the mapping table. This is done by the so called *Eviction Pointer* that points to the mapping table entry of the function that is evicted next. The two pointers can be implemented as registers and their behaviour is described as:

- **Write Pointer (WP)**: Points always to the address next to the end of the function that was inserted last.
- **Eviction Pointer (EP)**: On eviction of a function the pointer is moved to the next mapping table entry. Thereby, always the oldest function is selected.

If during function load the WP reaches the scratchpad address of the function marked by the EP, the function has to be invalidated. This is done by invalidating the mapping table entry and the lookup table entry selected by the EP.

Example. The Figure 3.13 shows the usage of the WP and EP on eviction. The functions were added to the D–ISP in the following order: a, b, c. Therefore, the EP marks the oldest entry in the memory, which is function a. The WP selects the next address after the end of function c that was inserted last.

On activation of function d, which is not in the scratchpad memory, it has to be loaded. Because the WP reaches the end of the scratchpad memory during write it is reset to the beginning of the scratchpad. Therefore, the function d is wrapped around the end of the scratchpad memory, but it is sequentially addressable by the *fetch control*. While writing the fetch blocks

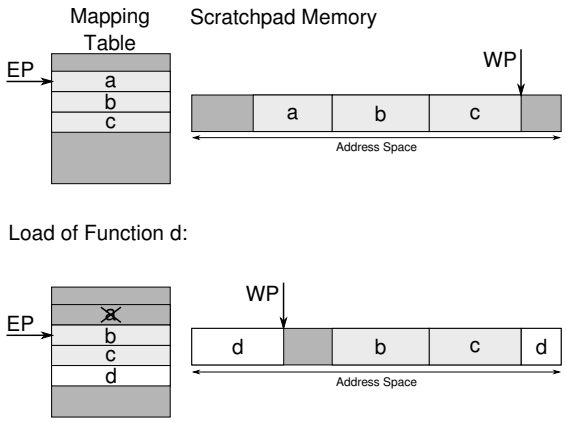


Figure 3.13: Example for the FIFO replacement policy using the Eviction Pointer (EP) and the Write Pointer (WP)

into the scratchpad the WP is compared with the scratchpad address of function that is selected by the EP. If both are equal the *content management* detected an overwrite of a function. On overwriting of the first fetch block of function a, the corresponding mapping and lookup table entries are invalidated. Both can be accessed by the EP, because the mapping table entry ID can be mapped directly to the lookup table. Then the EP is set to the next most function entry in the mapping table to mark the function that will be evicted next, which is function b. After loading function d into the scratchpad memory the WP is pointing to the next address after the end of d. ♦

As described on eviction a mapping table entry needs to be cleared, but also the scratchpad address of the next oldest function, which will selected by the EP on eviction, has to be obtained. Because both invalidating the mapping table entry of the evicted function and requesting the scratchpad address of the function to be evicted next have to be performed at once, the mapping table memory needs a second access port. Otherwise it could not be ensured that the function eviction is correctly performed. See the following example for the need of a second memory port for the mapping table memory.

Example. Assume the latency of off-chip memory access is same as the latency of a mapping table access, say n. Furthermore, function f has the size of one fetch block. The function k is loaded and will evict function f and g. Under this assumptions the *content management* is not able to handle the eviction of the function g that is overwritten right after function f correctly, if the address of function g is not requested immediately from mapping table on the eviction of f:

$$\begin{aligned} \text{cycle } x \text{ (load block 1 of k and eviction of f):} \quad & [f_1, g_9, h_{10}]_{20}^{FIFO} \xrightarrow{k_3} [g_9, h_{10}, \hat{k}_1]_{20}^{FIFO} \\ \text{cycle } x+n \text{ (load block 2 of k and eviction of g):} \quad & [g_9, h_{10}, \hat{k}_1]_{20}^{FIFO} \xrightarrow{k_3} [h_{10}, \hat{k}_2]_{20}^{FIFO} \end{aligned}$$

In cycle x the function f is evicted. On cycle $x+n$ the next fetch block is obtained from the memory, which will overwrite the first block of function g. To successfully recognize eviction the *content management* needs in this cycle the scratchpad address of function g. This is only possible, if this address was requested from the mapping table in cycle x.

If the mapping table has only one memory port it could be either used for invalidation of the mapping of function f or for the address request for function g. To allow a correct eviction of

function g, its address has to be requested. Then the invalidation of function f is delayed. In this case the eviction can be detected correctly, but the invalidation has to be buffered. Due to the fact that it could be the case that every function in the scratchpad has a size of one block, an unknown number of invalidations have to be buffered. This would result in an high hardware amount to store all delayed function invalidations. Also the worst-case timing is affected, since all pending invalidations have to be performed subsequent to the function load.

Therefore, the invalidation and the request for the address of the function to be evicted next has to be performed parallel, if the mapping table has the same latency as the off-chip memory. Thus, a second port to the mapping table memory is necessary. ♦

The implementation of content management for the **stack-based** replacement policy is much like as the FIFO replacement policy with the difference that evictions can take place in two directions. So the stack-based replacement policy is implementable by using two write and two eviction pointers instead of one for write and eviction, that FIFO needs. This is because for call and return different write and eviction pointers are necessary. On call the *Call Write Pointer* is used. Its behaviour is equal to the WP of the FIFO policy. To detect an eviction on call a *Call Eviction Pointer* is needed. It points to the function that will be evicted next on call, by selecting the corresponding mapping table entry. On function return two additional pointers are employed: the *Return Write Pointer* and the *Return Eviction Pointer*. The following description gives a deeper insight on how these pointers are used:

- **Call Write Pointer (CWP)**: Points always to the address next to the end of active function.
- **Return Write Pointer (RWP)**: Points always to the address before the beginning of the active function.
- **Call Eviction Pointer (CEP)**: On return it selects the deactivated function. On call it remains unchanged, if no eviction is necessary. If a function has to be evicted on call, the function with the largest stack distance to the activated one is the callee of the evicted function and will be selected. By design the callee of the evicted function is the next most entry in the mapping table, such that the CEP only needs to be incremented.
- **Return Eviction Pointer (REP)**: On call it is set to the called function. If no function is evicted on return, it remains unchanged. Otherwise it is set to the caller of the evicted one, which is now the function with the largest stack distance to the active one. The caller of the evicted function is reachable by simply decrementing the REP and thus selecting the previous entry in the mapping table.

Because of the fact that on return the functions are put in the memory onto the left side of the currently active function, the RWP points to the address before beginning the currently active function. It has to be decreased on write and thus the function is loaded in reverse order. Therefore, the stack-based replacement policy needs to know the function size before loading. This requires the usage of the function size determination by FLE instructions, in which the size of the function is determined by decoding the first fetch block. Knowing the function size, the function is written from end to beginning into the scratchpad memory. By comparing the RWP with the end address of the function that is to be evicted next, which is marked by the REP, the overwriting of a function is detected. For calls the function eviction is done like for FIFO, by comparing the CWP and CEP. When a eviction is detected the mapping table entry and lookup table entry marked by the CEP/REP is invalidated and the eviction pointer is altered to select the function with the largest stack distance to the active one. The latter is done by incrementing the CEP or decrementing the REP, respectively.

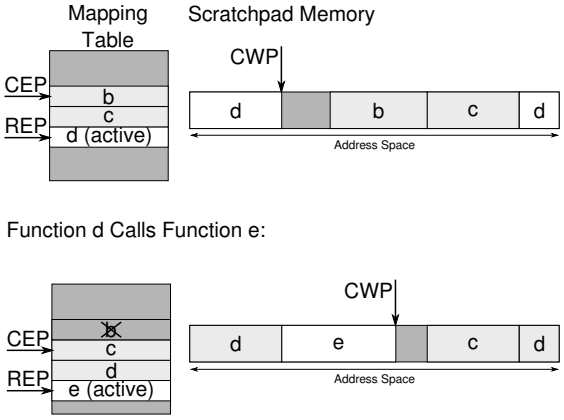


Figure 3.14: Example for the stack-based replacement policy on call using the Call Eviction Pointer (CEP) and the Call Write Pointer (CWP)

In the following an example clarifies the usage of these pointers for function evictions on call and return. Therefore, Figures 3.14 and 3.15 show the eviction pointers selecting entries of the mapping table and the write pointers marking the write position in the scratchpad memory.

Example. Assume the following function call order $a \xrightarrow{calls} b \xrightarrow{calls} c \xrightarrow{calls} d \xrightarrow{calls} e$. The Figure 3.14 shows the actions of the *content management*, if the active function d calls function e. During function write the CWP is increased until it reaches the start of function b. Then the *content management* detects by the use of the function start address selected by the CEP that function b is evicted. It invalidates the mapping of function b and sets the CEP to the successor entry of b, which is function c. So the next function that will be evicted is function c, which has the largest stack distance to the function that is loaded. On completion of the load of function e, the CWP points to the next most address after e. This behaviour is similar to the FIFO replacement. In addition to this the REP will be set to the activated function, which is e. This means that on return function e will be evicted next. Due to the call hierarchy this will happen, if function c returns to function b, which is not in the scratchpad. All other returns to reach function b will hit the scratchpad, except if another function is called in the meantime, but then this function is the one to be evicted next on return and the REP was updated to select this function.

The second example shows eviction in the case of return. In Figure 3.15 the execution of function b is completed and the caller function a is reactivated. Before the load of a the RWP points to the address in front of function b. To determine the size of a, the D–ISP requests the first fetch block from the memory. Then the fetch blocks of the function are requested from the memory in reverse order, while the RWP is decreased with each fetch block that is loaded. So the function is correctly loaded from end to beginning into the scratchpad. If the RWP points to the end address of the function that is selected by the REP, this function is evicted. In this example this is function d, which has the largest stack distance to the function that is activated. The *content management* invalidates the table entries of d and sets the REP to the predecessor entry of d. So the next function that will be evicted is function c. The last fetch block that is written is the first block of the function a. After handling the loading of function a the CEP selects function b, which will be evicted, if function a calls a function that is not in the scratchpad. So siblings in the call tree evict each other on call. The evaluation in Section 6.2, the impact of this possibly unnecessary eviction shown in detail.

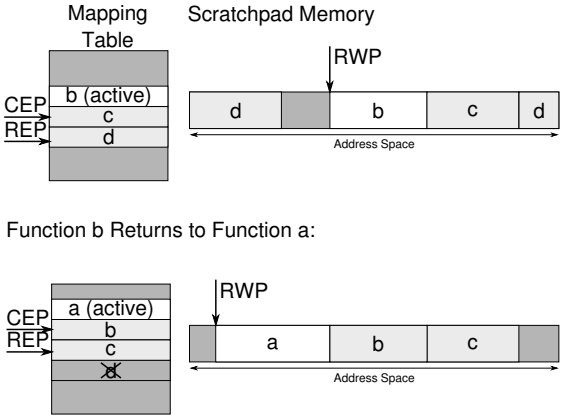


Figure 3.15: Example for the stack-based replacement policy on return using the Return Eviction Pointer (REP) and the Return Write Pointer (RWP)

Notice that the function load on return has to be done in reverse ordering, since otherwise the order of function evictions on return does not correspond to their stack distance. Then the replacement strategy implemented by the REP is not applicable. Assume that the function a is much larger than shown in Figure 3.15 and it also evicts function b and c. So when loading function a from beginning to end into the scratchpad the functions are overwritten in the following order: b, c and then d. But due to function b is the function with the lowest stack distance, it is not supposed to be evicted first. However, it can be evicted depending on the size of function a. But from the replacement policy implementation's point of view the next function that has to be evicted is function d, independently of the size of function a. Thus, the loading process has to be organized such that function d will be evicted first and then function c followed by function b, otherwise the described implementation with the REP cannot be used. This eviction order is enforced by the reverse load order. ⧫

The mapping and lookup tables are accessed in the stack-based policy in the same way as for FIFO, because either the CEP or the REP is used, which is decided early on transition to the **Table Lookup** state, see Figure 3.12. So only the CEP or the REP needs to be updated during function load. Thus, the stack-based replacement policy has the same requirements to the mapping and lookup table memory as the FIFO replacement policy.

For the **LRU** replacement policy the dynamically changing eviction order that depends on the function activation history renders the implementation in the D–ISP's *content management* as very complex. Furthermore, the fact that the size of the functions vary in a large range complicate the implementation.

If all functions would use a unified size in the scratchpad memory, LRU can be implemented using age counters for each function block. Such implementation is comparable to LRU implementations in caches. But as discussed in Section 3.2.1, equally sized memory blocks are not a valid choice, since the function sizes differ in a large range and the maximum function size may not be known on system design time. Another drawback is that the scratchpad will suffer a high memory fragmentation, due to unused parts in the memory blocks.

As outlined in Section 3.2.3, a *content management* in which a function is stored in multiple memory blocks and that is still implementable in hardware has to underly the restrictions: (1) a function will not be relocated after its load and (2) a function has to be stored in continuous

memory blocks. Unfortunately, when applying both restrictions a true LRU replacement policy cannot be implemented in the D–ISP *content management*.

If the scratchpad would be allowed to relocate functions, e.g. to merge small free memory regions such that a larger function fits, it has to keep track of free memory parts, determine the best memory layout, move functions, and update mapping table entries for relocated functions. By the complexity of these tasks, that are even challenging in software, it is not suitable to implement such approach in hardware. Since the time for the function relocations has also to be taken into account, a tight timing analysis of the D–ISP is impeded.

Allowing the fragmentation of functions in the memory, instead of function relocation, will also render the D–ISP implementation very complex, because all fragments of a function have to be maintained by the controller. The *content management* has to consider all scattered function parts on function eviction and mark them as free. On function load the free scratchpad memory parts have to be used to store the new function. Furthermore, the use of fragmented function mappings would add more pressure to the fetch path of the processor, because the address calculation in the *fetch control* is much more complicated for fragmented mappings. For example the implementation of the *fetch control* has to check an unknown number of blocks for the affiliation to the currently active function, such that either the implementation is very hardware intense or the fetch latency of the scratchpad would be increased. To mitigate the complexity of the *fetch control* the memory layout has to be optimized frequently to reduce the fragmentation, which will on the other hand escalate complexity of the *content management* and also adds unpredictability to the D–ISP's timing behaviour.

For these reasons the LRU replacement policy could not implemented in the D–ISP *content management*. Anyhow, the LRU replacement strategy is analysable [Reineke et al., 2007] and is known for the lowest miss rate [Al-Zoubi et al., 2004]. Thus, it will be used as baseline for the comparison of the other implemented replacement policies.

Processor Control

The TriCore processor comes with Core Special Function Registers (CSFR), which are used e.g. for memory management or memory protection [TriCore, 2007a]. These registers are also implemented in the CarCore architecture e.g. for scheduling control. For the D–ISP the handling of the CSFRs was extended to allow controlling the behaviour of the D–ISP. The following functions where implemented for the D–ISP:

- de-/activate D–ISP
- D–ISP start trigger
- flush scratchpad content

The activation and deactivation of the D–ISP allows the application programmer to decide if the D–ISP is used in different phases of an application. For example in the application's initialization phase the D–ISP is not needed, because this phase is usually not timing critical and due to its unique execution it may not benefit from the dynamic content management. Also the deactivation could be used on interrupts or operating system calls, that would disturb the content of the D–ISP. By the use of a start trigger the D–ISP can be activated by the call of a selected function. This eases the handling of (re-)activation of the D–ISP e.g. after initialization phase or an interrupt. Furthermore, the D–ISP *content management* supports a content flush, which is necessary on task switches. Then a started or reactivated task will get an empty scratchpad memory on its activation, which is of importance for the WCET analysis. The need for a content flush will be discussed in detail by Section 5.2.

Chapter 4

Analysis of Instruction Memories

The analysis of the memory content of caches and other memories like the D–ISP and so the determination of the memory access latency is crucial for a correct and precise WCET analysis. The focus and the contribution of this chapter is low-level analysis for the D–ISP, which is a step of the WCET analysis of the system as introduced in Section 2.6. But before the D–ISP analysis is described, the analysis for static memories and their content assignment and the topic of memory content analysis for instruction caches are discussed. These two sections are mainly known work, but they describe the analysis of the memories, which will be used for a comparison to the D–ISP in the evaluation of Chapter 6. The Section 4.1 provides the WCET-aware content selection for instruction memories and describes methods to optimise the WCET estimate by using static scratchpad memories. The content analysis for instruction caches with different replacement policies is described in the Section 4.2. This section lays the foundation for the content analysis of the D–ISP. The analysis of D–ISP described in Section 4.3 enables the WCET analysis for a system with the D–ISP and thus is the major contribution of this chapter. The proposed analyses use *abstract interpretation* for LRU and *collecting semantics* for FIFO and the stack-based replacement policy. The behaviour and implementation of the D–ISP replacement policies were introduced in Section 3.2 and 3.3, respectively.

4.1 Analysis and Assignment of Static Scratchpads

In the low-level analysis of static scratchpad memories for each memory object it is known if the memory accesses can be handled by the scratchpad or not, independently when in the application the memory object is accessed. However, the memory objects may be stored in multiple locations resulting in different access latencies. To assign the correct memory latency on access of a memory object the memory analysis has to be aware of the memory object's location. Then it is possible to add the corresponding memory access cost to the memory object during the low-level analysis. The static assignment of the memory objects results in their division to different memory access latencies classes, i.e. either the fast scratchpad is accessed or the rest of the memory hierarchy handles the memory access.

4.1.1 Assignment of Static Memory Content

The main challenge of the efficient usage of static scratchpad memories in hard real-time systems is the assignment of the memory objects to the static scratchpad. If this is done, the memory analysis merely has to categorize the accesses to the memory objects. The assignment of the

memory objects to the fast but small scratchpad can be done by different criteria: energy consumption [Wehmeyer and Marwedel, 2006], performance [Angiolini et al., 2004], and impact on the WCET estimate [Falk and Kleinsorge, 2009]. Since in hard real-time systems the reduction of the WCET estimate is most important, a WCET-aware static scratchpad assignment will be considered in this work. But it is also possible to employ other criteria and also lower the WCET estimate of the system as [Wehmeyer and Marwedel, 2004] shows. In the following the focus is on static instruction memories only, but similar approaches for static data memories are also proposed as e.g. by Suhendra et al. [2005].

To assign instruction memory objects their sizes and their execution cost in the worst case (for the cases they are executed from the scratchpad and for the case they are not) have to be calculated. Memory objects can be single instructions, basic blocks, multiple continuous basic blocks, or even whole functions. If the memory object type is not restricted, the general term *snippet* is used in the following. The size of the snippet is given by its contained instructions and is denoted as $size_s$. To calculate the execution cost difference for a snippet the cost when executing it from the scratchpad memory and when executing it from the off-chip memory have to be obtained. If a snippet contains control flow changes, like for functions, its internal WCET-critical path has also be determined to correctly calculate a valid upper bound for its execution cost. Then the cost difference Δc_s of a snippet s can be calculated as follows:

$$\Delta c_s = (cm_s - cs_s) \tag{4.1}$$

With cm_s as the cost for executing s from off-chip memory and cs_s representing the cost when executing s from instruction scratchpad. The calculation of the different costs for the snippet requires for both memories a separate WCET analysis under consideration of the timing characteristics of applied memory.

Scratchpad Assignment as Solution of a Knapsack Problem

The assignment of the snippets can be formulated as a *0-1 knapsack problem* (see e.g. [Cormen et al., 2003]): Each snippet has a known size and a benefit if it is put into the scratchpad memory and the scratchpad memory has a limited size. Then by maximizing the benefit over all snippets of the program an optimal selection of the snippets that will be assigned to the scratchpad can be found.

The benefit of a snippet depends on how often it is executed on the WCET-critical path (WCP) of the application. So to calculate the number of times a snippet is accessed on the WCP a WCET analysis of the application is to be done. For the determination of the WCP refer to the related work on path analysis discussed in Section 2.6. Then the $benefit_s$ of a snippet s can be calculated by Equation (4.2) using the cost difference Δc_s (from Equation (4.1)) for the execution of the snippet from the different memories and the activation count aw_s of the snippet on the WCP as weight.

$$benefit_s = \Delta c_s \cdot aw_s \tag{4.2}$$

To find the optimal assignment for the scratchpad among all snippets that are executed on the WCP the overall benefit is to be maximised, but the scratchpad size has not to be exceeded. In literature such knapsack problems are often solved by linear programming as e.g. in [Wehmeyer and Marwedel, 2004]. The formulation of the knapsack optimisation problem of assigning snippets to the scratchpad memory as linear program (LP) [Luenberger and Ye, 2008]

Algorithm 4.1 Knapsack-based snippet assignment for the static instruction scratchpad

Input: Program: P ∧ Snippets: S ∧ Scratchpad size: size
Output: Assignment of snippets to scratchpad: x

```
x ← {0, ··· , 0} // Clear assignments
wcp ← calculate_WCET(P) // Calculate the WCET and deliver WCP
for ∀s ∈ S do
  calculate_snippet_benefit(P, s, wcp) // Calculate benefit_s
end for
x ← generate_and_solve_LP(S, wcp, size)
return x
```

in the linear program standard form is shown below:

$$
\begin{aligned}
\textit{maximise} \quad & \sum_{\forall s \in snippets} x_s \cdot benefit_s \\
\textit{subject to} \quad & \sum_{\forall s \in snippets} x_s \cdot size_s \quad \leq size_{ISP} \\
\textit{with} \quad & x_s = \begin{cases} 0 & if \text{ snippet } s \text{ is located in off-chip memory} \\ 1 & if \text{ snippet } s \text{ is located in scratchpad memory} \end{cases}
\end{aligned}
\tag{4.3}
$$

The assignment of a snippet s is denoted by x_s. The assignments for all snippets is found by solving of the linear program (4.3). The Algorithm 4.1 shows the steps needed to obtain the scratchpad assignment x using linear programming. After calculating the WCET estimate and determining the WCP the benefit for each snippet is calculated. Then to obtain the assignment the linear program for the knapsack problem is generated and solved.

Linear programs are known to find the optimal solution for *0-1 knapsack problems* [Cormen et al., 2003], also in this case the optimal set of code snippets is assigned to the scratchpad memory. But only snippets that are on the initial WCP are taken into account during the assignment to minimize the WCP's execution time, all other paths are ignored. This is expressed by the weight aw_s in Equation (4.2). Unfortunately, by optimizing the WCP another path may become the worst-case execution time critical path. So it is possible that the overall WCET of the application is not as much affected as calculated by the LP. In the worst case the WCET estimate may not be reduced by assignment of snippets to the scratchpad. Anyhow, the solution of the LP is safe, because it has no negative impact on the WCET estimate[1], but it does not guarantee its improvement.

Example. Consider a control flow graph with two parallel control flows A (with snippets s_1 to s_3) and B (s_4 to s_6) of equal cost as depicted in Figure 4.1. For each snippet the cost c is given that contains of the cost for the execution of the snippet in scratchpad and off-chip memory. On WCET calculation the analysis has to select one of the paths with equal cost as WCP, say path A will be selected. If assigning the snippets s_1 to s_3 to the scratchpad memory, the cost for path A can be reduced to half of its initial cost. But on recalculation of the WCET the path B will be the WCP. Then the WCET is not reduced by using a static scratchpad and selecting its content by a knapsack LP solution. ♦

Due to the fact that the LP considers only snippets on the precalculated WCP, the knapsack formulation cannot deliver the optimal assignment, if the WCP changes by solving the problem.

[1] With the assumption that the assignment of an arbitrary snippet is free of any penalty.

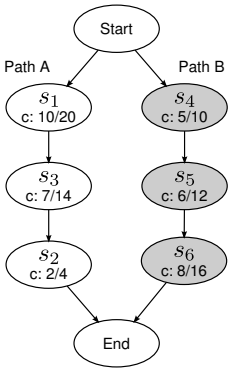


Figure 4.1: Example for a non-optimal knapsack solution with linear programming (The cost of the snippets s_1 to s_6 is depicted by $c : cs/cm$, e.g. s_1 with $cs = 10$ and $cm = 20$.)

As shown in the example above this is the case for applications with disjoint paths that can dominate the WCET.

WCP-sensitive Scratchpad Assignment

To provide an optimal assignment of snippets to the scratchpad in terms of a minimum WCET estimate the changes of the WCP that are caused by the assignment of a snippet to the scratchpad have to be taken into account. In [Suhendra et al., 2005] a WCP-sensitive ILP formulation for the assignment of data objects to a scratchpad is proposed. Suhendra et al. present a scheme for a control flow that is free of loops (i.e. there are no back-edges) represented by a *directed acyclic graph* (DAG) in which the cost of all possible paths and the benefits of all assigned data objects is considered. By representing a loop body as a virtual node for which its cost is multiplied by the maximal number of iterations, loops can be constructed and embedded using the formulation proposed by Suhendra et al. Thus it can be applied for any general program without recursion. This approach was adapted by [Falk and Kleinsorge, 2009] to model an optimal WCET-aware assignment of basic blocks to a scratchpad. In the following it is discussed for the simplifying assumption that the assignment of a snippet does not affect the size and the precalculated timing of the snippet itself and of other snippets. This constraint has to be relaxed later in Section 4.1.3 in which the approach of Falk and Kleinsorge [2009] is discussed in all its facets. In the following the general idea of the approach of Suhendra et al. [2005] and Falk and Kleinsorge [2009] is described.

Assume a DAG G (see [Jungnickel, 1999]) representing the control flow of a code part that consists of basic blocks and has no back-edges, e.g. a loop body. Furthermore, this DAG has one entry and one exit node. The DAG is defined as follows by $G = (V, E)$. With V as the set of basic blocks and E as the set of all edges in the control flow represented by the DAG. If the DAG represents a loop body, the loop back-edge from the exit to the entry node is omitted and thus not contained in E, i.e. $(exit, entry) \notin E$.

Then the **worst-case execution cost** of each basic block (denoted as w) within G can be modelled by the execution cost of the block itself (denoted as cx) and the worst-case execution cost of its succeeding basic block. The worst-case execution cost of the DAGs exit node is determined by its execution cost only:

$$w_{exit} = cx_{exit}$$

Where w_{exit} denotes the worst-case execution cost and cx_{exit} the execution cost of the node $exit$. According to [Falk and Kleinsorge, 2009] the worst-case execution cost of all other basic blocks in the DAG can be calculated by:

$$\forall i \in V \setminus \{exit\} : \forall (i,j) \in E : w_i \geq w_j + cx_i$$

Notice that for a basic block i with different successor nodes all paths are represented separately. Therefore, the worst-case execution cost of i is depending on all possible control flows that lead to the basic block i. So WCP switches during snippet assignment are taken into account.

To improve the precision of the cost estimation (cx) for the basic blocks compared to the definition from [Falk and Kleinsorge, 2009] in the following also the edge on which each basic block is left is considered.

Example. Assume the basic blocks u, v, and w, for which u contains a conditional jump. If the jump is taken w is executed, else v (q.v. Figure 4.2(c) on page 92). Further assume that the processor does not use any branch prediction. Then the processor's timing model needs only to charge the jump penalty, which is required to calculate and forward the new program counter to the instruction fetch stage or flush the prefetched instructions, if the jump is taken, represented by the edge (u, w). For a timing model that takes this effect into account the cost of a basic block depends on its outgoing edge in the control flow graph:

$$c_{(u,v)} < c_{(u,w)}$$

If the timing model of the processor does not distinguish both cases, the maximum cost of all possible cases needs to be considered:

$$c_u = \max\left(c_{(u,v)}, c_{(u,w)}\right)$$

This overestimates the case in which the conditional jump is not taken, represented by the edge (u, v) in the control flow graph. So by distinguishing the way how a basic block is left its cost can be determined more precisely. ♦

Hence, preciser cost for each basic block can be calculated by assigning the cost to the outgoing edge of each basic block ($cx_{(i,j)}$), as shown by the Equation (4.4) below.

$$\forall i \in V \setminus \{exit\} : \forall (i,j) \in E : w_i \geq w_j + cx_{(i,j)} \tag{4.4}$$

To model a **loop** L or a **function** F a virtual node v is created. The virtual node encapsulates the whole control flow of the loop body respectively the function. The entry point of the control flow encapsulated by v is denoted as $entry(v)$ and the exit node is denoted as $exit(v)$. The predecessor node of v is i (i.e. it is the node that is executed before loop L or that calls function F, respectively) and the node j is the successor node of v (i.e. the node after the loop L or the node to which the function F returns after execution, respectively). Then a path $< i, v, j >$ takes the encapsulated control flow of L or F into account.

As already described above the control flow is modelled as a DAG and at first only the control flow of v is taken into account, independently of any control flow it is embedded in. At this stage any connections (to i and j) are unknown. Since the basic block cost is assigned to its outgoing edges, the execution cost of the exit node $exit(v)$ cannot be determined at this point. Thus it is set to 0:

$$w_{exit(v)} = 0 \tag{4.5}$$

The cost for the exit node is assigned to the loop conserving edge ($e_{vcons} = (exit(v), entry(v))$) and to the loop/function exiting edge ($e_{vexit} = (exit(v), j)$). The cost of these edges is charged

at the virtual node respectively at the connection to its successor node. The worst-case execution cost of all basic blocks within the loop body L or the function F is calculated using Equation (4.4).

To connect the control flow of v to the rest of the control flow the cost variable c_v for the internal control flow of the virtual node is introduced. Its definition depends on if v represents a function or a loop body. In both cases the dominating term is the worst-case execution cost of the entry node $w_{entry(v)}$:

$$c_v = \begin{cases} w_{entry(v)} \cdot b_L + cx_{(exit(v),entry(v))} \cdot (b_L - 1) & if \text{ v is a loop with loop bound } b_L \\ w_{entry(v)} & if \text{ v is a function} \end{cases} \tag{4.6}$$

For a loop this cost is multiplied with maximum number of loop iterations of L given by the loop bound b_L. In addition to that the cost for the loop conserving edge ($e_{vcons} = (exit(v), entry(v))$) is taken into account $b_L - 1$ times, representing the number of times the loops exit node is executed and next loop iteration is started. If v is a function, no loop bounds have to be considered as shown in the second case of Equation (4.6).

The virtual node is connected to its neighbour nodes i and j as follows:

$$w_i \geq w_v + cx_{(i,entry(v))} \tag{4.7}$$

$$w_v \geq w_j + cx_v + cx_{(exit(v),j)} \tag{4.8}$$

Notice that for the virtual node v the cost of the exit node considering the loop/function exit edge ($e_{vexit} = (exit(v), j)$) is considered when connecting the virtual node to its successor j.

If there are multiple predecessors nodes $Pred_v = \{i_0, \cdots i_n\}$ for v, the Equation (4.7) is to be generated for each predecessor. Analogous to this multiple successor nodes $Succ_v = \{j_0, \cdots j_m\}$ for v are considered by building Equation (4.8) for each successor node:

$$\forall i \in Pred_v : w_i \geq w_v + cx_{(i,entry(v))}$$
$$\forall j \in Succ_v : w_v \geq w_j + cx_v + cx_{(exit(v),j)}$$

Using the equations above the control flow of the loop L is encapsulated by the virtual node v that replaces L in the representation of the control flow in which L is embedded. By an iterative process all loops in the application can be replaced by virtual nodes from bottom up. A function F is also replaced by a virtual node to simplify the linear program, because F may be called from different places in the application. The transformation of functions into virtual nodes implies the assumption that the WCET and the WCET-critical path is independent of the call context of the function. If this is not the case, different call contexts of functions cannot be encapsulated by the same virtual node. For the simplification of the assignment of snippets to a scratchpad the different call contexts of a function are not distinguished, i.e. the worst-case context of the function is always assumed.

Hence, any control flow in the whole program (except recursion) can be modelled by the formulations shown above. The control flow information of the application then aggregates in the program's entry point with the worst-case execution cost of $w_{program\_entry}$.

To model the **assignment of the snippets** to the scratchpad every basic block belongs to exactly one snippet. The mapping of a basic block i to a snippet s is represented by a set of basic blocks that belong to one snippet (B_s):

$$\begin{aligned} & B_s = \{\cdots, i, \cdots\} \\ \forall i \in V, \exists s \in snippets : \quad & i \in B_s \\ \forall s \in snippets : \quad & B_s \neq \emptyset \\ \forall i \in V, \exists s \nexists t \in snippets, s \neq t : \quad & i \in B_s \wedge i \in B_t \end{aligned}$$

The cost of a basic block cx_i depends on the assignment status of the snippet s to which it belongs ($i \in B_s$). This is modelled by the following constraint, that charges the cost for the execution of basic block i from the scratchpad (cs), if the snippet s is assigned to the scratchpad ($x_s = 1$). Otherwise the cost for the execution of i from the off-chip memory (cm) is taken into account.

$$\forall i \in B_s, \forall s \in snippets : \forall (i,j) \in E : cx_{(i,j)} = cm_{(i,j)} \cdot (1 - x_s) + cs_{(i,j)} \cdot x_s \tag{4.9}$$

With these formulations it is possible to find the optimal assignment of the snippets (x_s) to minimise the WCET under consideration of changing WCPs caused by the assignment of snippets to the scratchpad. The resulting LP is defined as follows:

$$\begin{aligned} &minimize && w_{program\_entry} \\ &subject\ to && \sum_{\forall s \in snippets} x_s \cdot size_s \quad \leq size_{ISP} \\ &with && x_s = \begin{cases} 0 & if \text{ snippet } s \text{ is located in off-chip memory} \\ 1 & if \text{ snippet } s \text{ is located in scratchpad memory} \end{cases} \end{aligned} \tag{4.10}$$

The objective function of the LP minimizes the worst-case execution cost of the program's entry point. When applying the Equations (4.4) and (4.5) for the whole program and replacing all loops and functions by virtual nodes as shown in Equations (4.6) to (4.8) the full control flow of the application is represented in this worst-case execution cost value ($w_{program\_entry}$). The minimizing of this cost by the objective function calculates the snippet assignment under consideration of the size of the scratchpad and changes of the WCET-critical path. This is in contrast to the knapsack formulation shown in LP (4.3) that searches for the optimal assignment of the snippets only along the previously calculated WCP.

The calculation of the optimal assignment by minimising the worst-case execution cost of the application is completely different to the WCET calculation using the IPET (q.v. Section 2.6) that maximizes the flow through the control flow graph to obtain the WCP. The difference in the LP (4.10) is that the loop bounds are taken into account by statically charging the cost of the maximum number of loop iterations (see Equation (4.6)) instead of finding the maximum flow through the loop which is upper-bounded by a maximal iteration count. Thus it is not possible for the LP (4.10) to find or model a WCP of a loop that contains of different WCET-critical paths for different iterations of the loop. Furthermore, flow constraints provided by the user or a high-level analysis that describe different execution paths within one loop (e.g. for an *if-then-else* block in the loop body of which for half of all iterations the *then*-branch is taken and for the rest of the iterations the *else*-branch is executed) or a function cannot be modelled. However, when assuming the WCP of a loop or function is the same for all iterations or call contexts, the LP (4.10) safely finds the WCP of the application.

This is also due to the greater-equal operator in Equation (4.4), which ensures that the maximum cost of a basic block is not underestimated (by e.g. taking the cheaper path or successor into account). Hence, as also stated in [Falk and Lokuciejewski, 2010], which embeds the work of Falk and Kleinsorge [2009] into a WCET-aware compiler (see Section 2.6.3), the shown formulation delivers the optimal WCP-sensitive snippet assignment for a scratchpad of a given size.

The Algorithm 4.2 shows how the WCP-sensitive algorithm can be embedded into an analysis tool. Notice that in contrast to the knapsack formulation, as shown in Algorithm 4.1 the WCET does not need to be calculated before the scratchpad assignment is determined. Instead, the WCP is calculated by the LP while assigning the snippets to the scratchpad.

Algorithm 4.2 WCP-sensitive snippet assignment for the static instruction scratchpad

Input: Program: P ∧ Snippets: S ∧ Scratchpad size: size
Output: Assignment of snippets to scratchpad: x

```
x ← {0, ··· , 0} // Clear assignments
for ∀s ∈ S do
    calculate_bb_cost(P, s, ISP) // Calculate ∀i ∈ B_s : cs_i
    calculate_bb_cost(P, s, MEM) // Calculate ∀i ∈ B_s : cm_i
end for
x ← generate_and_solve_WCP_sensitive_LP(P, S, size)
return x
```

Code Relocation

After assigning the snippets to the scratchpad memory the corresponding code needs to be relocated to the scratchpad. This can be done by linking the code to another memory region or explicitly copy the code on system initialisation. Because the alignment of the code is crucial for the timing characteristics of the application[2], the relocated code is to be aligned in the scratchpad as it was in the off-chip memory. Furthermore, the alignment of the remaining code in the off-chip memory has not to be changed by relocation of the code that is put into the scratchpad. Hence, it is recommended to copy the assigned code during system initialisation to the scratchpad and leave the off-chip memory content unchanged.

To reach the code in the scratchpad jump and call target addresses need to be updated. This can be done by relinking the application (for functions only) or by altering the application code or the executable file. The details of mapping the selected code snippets into the scratchpad will be discussed later in Section 5.2.

4.1.2 Function-Based Assignment

A function-based assignment of the static scratchpad content is rather coarse grained, but the most comprehensible assignment strategy for the application programmer. A static scratchpad that only contains complete functions is further denoted as **FS–ISP**. The knapsack-based and the WCP-sensitive assignment approach for the FS–ISP is introduced in the following.

Knapsack-based Function Assignment

To find the optimal assignment for the FS–ISP the knapsack-based approach needs to obtain the benefit for relocating a function into the scratchpad. Therefore, the calculation of the WCP of all functions and the cost of all containing basic blocks is necessary. So depending on the WCP of a function f (denoted as WCP_f) the benefit of putting f into the FS–ISP can be calculated as follows:

$$benefit_f = \left(\sum_{\forall (i,j) \in E_{CFG}, i \in B_f} \Delta c_{(i,j)} \cdot aw_{(i,j)} \right) \cdot iv_f \tag{4.11}$$

With E_{CFG} as the edges in the control flow graph of the program. Since the assignment of the cost for a basic block is more precise, if the outgoing edge of the basic block in the control flow

[2]For example this is the case for a host processor that supports instructions of different length and is sensitive to the alignment of the instructions. Refer to Zhao et al. [2004; 2005] for a quantification of the impact of an alignment-aware code positioning on the WCET.

graph ($(i,j) \in E_{CFG}$) is taken into account, the benefit calculation uses $\Delta c_{(i,j)}$ instead of Δc_i as it was proposed in Equation (4.1) for a snippet.

Notice that no cost of the functions that are called by the function f is assigned to the benefit of f. Nevertheless, to determine the correct WCP of the function f the WCET analysis takes any function calls of f into account. The term $aw_{(i,j)}$ represents the activation count of the basic block i on the WCP_f with j as successor node. B_f is the set of basic blocks that belong to the function f. The Number of invocations of this function on the WCP of the whole program is taken into account in the calculation of the benefit by the term iv_f.

The knapsack formulation of the assignment problem as linear program is similar to the general case shown in the linear program (4.3):

$$
\begin{aligned}
maximise \quad & \sum_{\forall f \in functions} x_f \cdot benefit_f && (4.12)\\
subject\ to \quad & \sum_{\forall f \in functions} x_f \cdot size_f \quad \leq size_{ISP} \\
with \quad & x_f = \begin{cases} 0 & if \text{ function } f \text{ is located in off-chip memory} \\ 1 & if \text{ function } f \text{ is located in scratchpad memory} \end{cases}
\end{aligned}
$$

To determine the set of the selected functions the general Algorithm 4.1 can be used with the modification that functions instead of snippets are used. Because the number of functions is usually small in common applications, compared to the number of basic blocks or instructions, the generated LP to find the assigned functions will be of low complexity, since only one variable per function needs to be found.

The relocation of the assigned functions can be either done by relinking the application or by copying the selected code into the scratchpad using a startup routine and adjusting the target addresses of the affected calls.

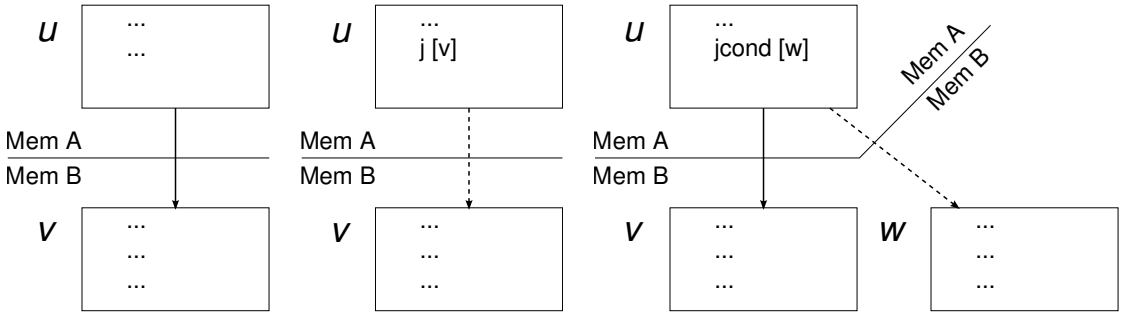
WCP-sensitive Function Assignment

To support functions in the WCP-sensitive assignment the basic block cost function, see Equation (4.9), needs to take its membership to a function into account. Therefore, the mapping of the basic blocks to the snippets B_s needs to be adjusted, such that every snippet represents a function that can be assigned to the FS–ISP. Since the control flow of the program is already modelled by LP (4.10), no further changes need to be applied.

As for the knapsack-based assignment, the selected functions can either be relocated to the scratchpad by the linker or by a startup routine that directly copies the code of the functions to the scratchpad.

4.1.3 Basic-Block-Based Assignment

Assigning the scratchpad memory on granularity of basic blocks (denoted as **BBS–ISP** in the following) allows a better utilisation of the memory and a more precise optimisation of the application's WCET estimate by selecting dominating hot spots, like a specific path in a loop. But the assignment of basic blocks to a scratchpad is complicated, because their connection in the control flow graph needs to be preserved. To keep the basic blocks connected after their relocation it is required to alter the assigned blocks and also some of their neighbours in terms of adding or adjusting connecting jumps to preserve the application's control flow. These adjustments may imply penalties for the selected blocks. Thus the basic-block-based assignment needs to consider the structure of the control flow and the specific penalty implications when a relocating



(a) Continuous addressing (b) Unconditional jump (c) Conditional jump & continuous addressing

Figure 4.2: Scenarios for the assignment of consecutive blocks to different memories

basic blocks to the scratchpad. But before discussing the effects of adding jumps to preserve the control flow, the simplified case for the assignment of the BBS–ISP will be described.

Knapsack-based Basic Block Assignment Without Additional Penalties

When ignoring any penalties to preserve the control flow related effects by relocating basic blocks into the scratchpad memory space, the benefit of a basic block i ($benefit_{(i,j)}$) when assigning it to the BBS–ISP can be calculated as follows:

$$\begin{aligned} benefit_{(i,j)} &= (cm_{(i,j)} - cs_{(i,j)}) \cdot aw_{(i,j)} \\ &= \Delta c_{(i,j)} \cdot aw_{(i,j)} \end{aligned} \tag{4.13}$$

For a precise estimation of the cost of basic blocks the benefit calculation takes the edge in the control flow graph into account on which basic block i is left, which is $(i,j) \in E_{CFG}$ with E_{CFG} as the set of all edges in the control flow graph. A WCET analysis has to be performed to obtain the initial WCP and the execution cost of every basic block for executing it from the scratchpad (cs) and from off-chip memory (cm). The activation count of a basic blocks i on the WCP is denoted by $aw_{(i,j)}$, with j as successor node of i on the WCP.

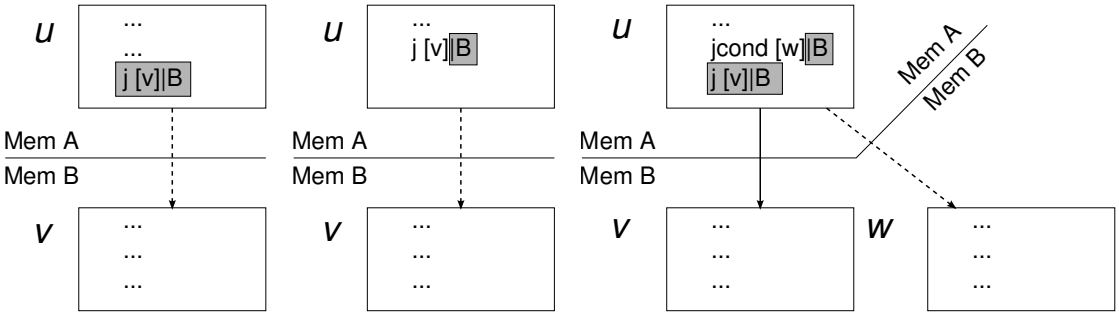
The LP formulation for the basic block assignment is similar to the general formulation shown in the linear program (4.3) with the difference that basic blocks instead of abstract code snippets are used:

$$\begin{aligned} maximise \quad & \sum_{\forall (i,j) \in E_{CFG}, i \in basic\_blocks} x_i \cdot benefit_{(i,j)} \\ subject\ to \quad & \sum_{\forall i \in basic\_blocks} x_i \cdot size_i \quad \leq size_{ISP} \\ with \quad & x_i = \begin{cases} 0 & if \text{ basic block } i \text{ is located in off-chip memory} \\ 1 & if \text{ basic block } i \text{ is located in scratchpad memory} \end{cases} \end{aligned} \tag{4.14}$$

To determine the basic-block-based assignment the general Algorithm 4.1 can be easily adjusted to obtain the optimal set of assigned basic blocks.

Knapsack-based Basic Block Assignment Including Jump and Size Penalties

If connected basic blocks are assigned to the BBS–ISP, the additional cost of the transition on which the scratchpad is entered or leaved needs to be taken into account. Two main cases can



(a) Continuous addressing (b) Unconditional jump (c) Conditional jump & continuous addressing

Figure 4.3: Changes for consecutive blocks in different memories to preserve the control flow

be distinguished: a block in the other memory is reached by *continuous addressing* or by *jump*. The Figure 4.2 depicts those cases: in (a) the basic block v is reached from u by continuous addressing, in (b) an unconditional jump activates the basic block v, (c) shows a conditional jump that activates basic block w in case of jumping otherwise basic block v is activated.

By assignment of basic blocks to different memories, say A and B, the blocks have to be moved to the different memories, meaning a changed address space and also a different position within the memory. Thus, the connection of the basic blocks in the control flow needs to be updated to execute the application correctly. For the case that the basic block u is in memory A and basic block v respectively w is in memory B the basic block u needs to be altered to reach v and w which are in a different address space. In doing so it doesn't matter if the scratchpad is entered of left by the transition from u to v or from u to w, respectively.

For case (a) this is done by adding an additional jump to basic block v at the end of basic block u, as shown in Figure 4.3. For jumps, depicted in Figure 4.3(b), only the jump target address needs to be adjusted. This can also require the replacement of the jump instruction to reach the memory B, because the offset that is available in the original jump instruction may not be large enough to reach the address space of memory B. Jumps that leave a memory region may be more costly than the jump instruction that was replaced, e.g. by the necessity to add an extra instruction, which sets a register to the target section for a jump that needs a base register and an offset to reach the jump target in memory B. If a basic block contains a conditional jump (case (c)) an additional jump to reach v has to be inserted and the jump target to reach w has also to be changed. This is shown in Figure 4.3(c). Strictly this adjustment will break the definition of a basic block for u. Though, to keep the consistency in the equations and LPs in the following the term basic block is still used for blocks that had to be altered to preserve the control flow like block u. The changes that are required to reach the basic blocks in the different memories are depicted in Figure 4.3. The adjustments to preserve the control flow of the basic blocks may have different costs: for the cases (a) and (c) an additional jump has to be considered and for the cases (b) and (c) an already present jump has to be altered.

In the following the LP formulation for the assignment with respect to the jump penalty is given:

$$maximise \quad \sum_{\forall(i,j)\in E_{CFG}, i\in basic\_blocks} x_i \cdot benefit_{(i,j)} - (x_i \oplus x_j) \cdot jp_{(i,j)} \cdot aw_{(i,j)} \tag{4.15}$$

$$subject\ to \quad \sum_{\forall i\in basic\_blocks} x_i \cdot size_i^p \quad \leq size_{ISP} \tag{4.16}$$

$$with \quad x_i = \begin{cases} 0 & if \text{ basic block } i \text{ is located in off-chip memory} \\ 1 & if \text{ basic block } i \text{ is located in scratchpad memory} \end{cases}$$

$$jp_{(i,j)} = \begin{cases} jp_{ca} & if \text{ j is reached from i by continuous addressing} \\ jp_{js} & if \text{ j is reached from i by short jump} \\ jp_{jl} & if \text{ j is reached from i by long jump} \\ jp_{cl} & if \text{ j is reached from i by call} \\ 0 & otherwise \end{cases} \tag{4.17}$$

The formulation is similar to the simplified case shown in linear program (4.14) with the additional jump penalty term. This term adds an jump penalty $jp_{(i,j)}$ for consecutive basic blocks, connected by the edge (i, j) in the CFG, if either one or the other $(x_i \oplus x_j)$ is assigned to the scratchpad. Furthermore, the activation count $aw_{(i,j)}$ of the basic block i on the WCP, which also traverses j, is used in the jump penalty term, because the number of activations of i affects the overall penalty, if i either or j is assigned to the scratchpad.

The weight of the jump penalty $jp_{(i,j)}$ depends on how the blocks are connected in the code, as Equation (4.17) shows. The constant jp_{ca} denotes the additional cost of inserting a new jump at the end of the basic block that connects the basic block to its successor with continuous addressing to bridge the gap in the address spaces of the different memories. The two other constants jp_{js} and jp_{jl} define the additional cost for the case that the blocks were already connected by a jump. Depending on the underlying ISA it is possible that a short jump and long jump result in different timing penalties, when they are replaced by a jump that connects two different address regions (e.g. the target address needs to be calculated by additional instructions). Notice that for a conditional jump the two differently connected basic blocks are taken into account with their own jump penalty. This is ensured by checking all edges of the program for potential jump penalties in the LP (4.15). The jump penalty jp_{cl} is for the case that a basic block is reached by a function call. Depending on the used ISA special call instructions could be required to jump into a different memory address space. Such call instructions could use register indirect addressing and need additional instructions that store the function's address in a register. This additional effort is represented by the jump penalty term for calls[3]. For a return from a function no additional jump penalty is assumed due to the return address is obtained in full address width from the stack without an additional penalty.

The benefit and the penalty term of the linear program (4.15) can be subtracted, because both represent the same measurement unit, which is the number of saved or additionally needed clock cycles, when basic blocks are assigned. Notice that the calculation of the benefit ($benefit_{(i,j)}$) is left unchanged.

As depicted in Figure 4.3 the conservation of the original control flow on assigning basic blocks to different memories, does not only affect the timing of the basic block. Moreover the

[3]For the FS–ISP the call penalty is not considered, because the impact of this penalty for memory objects with a larger benefit/penalty ratio rather small. Thus, wrong decisions of the scratchpad assignment caused by the penalty will be less likely for the FS–ISP.

code of basic blocks has to be changed. This can result in an increased size of the basic block. So on assignment of a basic block it is possible that the size of this block or of its predecessor increases. Because the assignment of the basic blocks has to fit the available scratchpad size, the effect of changing basic block sizes for assigned blocks has to be considered[4]. Therefore, the size of the basic block i depends on the assignment of itself and the succeeding basic block: A basic block can only suffer the size penalty, if it is assigned but one of its successor block is not. This is modelled in the $size_i^p$ variable in Equation (4.16) that is defined as:

$$size_i^p = size_i \tag{4.18}$$
$$+ \sum_{\forall (i,j) \in E_{CFG}} \begin{cases} (x_i \wedge \neg x_j) \cdot sp_{ca} & if \text{ j is reached from i by continuous addressing} \\ (x_i \wedge \neg x_j) \cdot sp_{js} & if \text{ j is reached from i by (un)conditional short jump} \\ (x_i \wedge \neg x_j) \cdot sp_{jl} & if \text{ j is reached from i by (un)conditional long jump} \\ (x_i \wedge \neg x_j) \cdot sp_{cl} & if \text{ j is reached from i by call} \\ 0 & otherwise \end{cases}$$

The different size penalties sp define the number of bytes that are needed to insert an appropriate jump instruction and/or additional instructions to calculate the jump displacement. The same three different scenarios as for the jump penalty term (see Equation (4.17)) are distinguished. Notice that for each basic block the size penalties for every possible exit edge are considered. This is done by aggregating the effects of all possible out edges of the basic block in the control flow graph. Thus, for a basic block with a conditional jump, which is assigned to the scratchpad and its both successors are not, the size increase caused by both connective jumps is taken into account.

The usage of the jump and size penalty terms is derived from [Falk and Kleinsorge, 2009]. But the authors have a different view on the program under observation: Falk and Kleinsorge implement the basic block assignment within a compiler. Such that they are able to not only relocate basic block to different memories it is also possible to optimise the structure of the application. For example if a complete control flow between two basic blocks i and j is relocated to the scratchpad and the basic blocks i and j were connected by a jump, then the compiler is able to omit the jump and connect the blocks by continuous addressing. This optimisation reduces the execution cost of the basic block i, since no jump is necessary. Whereas the assignment of the basic blocks to the scratchpad that is described in this section is to be implemented in a post-link WCET analysis tool (see Section 5.2) that intends to minimize the changes in the application's executable file. Hence, any structural optimisation of the application is not taken into account. So the formulations of the penalty terms where adapted to fulfil the capabilities of the post-link analysis tool.

To model the *exclusive-or* operation ($\oplus$) in the LP (4.15) it needs to be reformulated. A proper reformulation of the *exclusive-or* operation for binary values i and j is shown in [Brown and Dell, 2007]. By creation of a new variable that contains the *exclusive–or* value of i and j, the *exclusive-or* operator is realized:

$$\begin{aligned} i \oplus j \mapsto iXORj;\ iXORj = \{0,1\} \\ iXORj \leq i + j; \\ iXORj \geq i - j; \\ iXORj \geq -i + j; \\ iXORj \leq 2 - i - j; \end{aligned}$$

[4]The sizes of basic blocks that are not assigned to the scratchpad may also change, but due to the "infinite" size of the off-chip memory, this effect is ignored when assigning the basic blocks to the scratchpad.

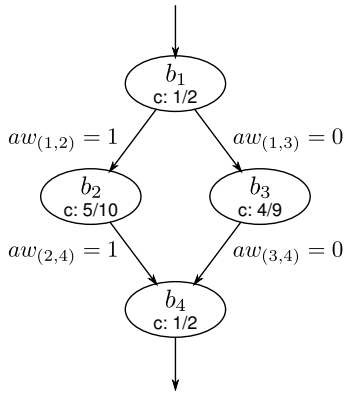


Figure 4.4: Example of a pathological case for the knapsack-based basic block assignment (The cost of the basic blocks b_1 to b_4 is depicted by $c : cs/cm$, e.g. b_1 with $cs = 1$ and $cm = 2$.)

For the logical *and-not* operation of Equation (4.18) a similar formulation can be found, by combining *and* and *not* according to [Brown and Dell, 2007]:

$$\begin{aligned} i \wedge \neg j \mapsto iANDNj;\ iANDNj = \{0,1\} \\ iANDNj \leq i; \\ iANDNj \leq 1 - j; \\ iANDNj + 1 \geq i + 1 - j; \end{aligned}$$

Using these inequations allows a sound formulation of the jump and size penalties for the linear program (4.15).

The following example shows a pathological case for which the knapsack based formulation for the assignment of a BBS–ISP delivers a worse WCET estimate, by assigning basic blocks to a scratchpad.

Example. Consider the control flow shown in Figure 4.4 with four basic blocks b_1 to b_4. The cost of the basic blocks are shown within each node: e.g. b_2 has the cost of 10 if it is executed from the off-chip memory and the cost of 5 when it is located in the BBS–ISP. A WCET analysis will identify the path $< b_1, b_2, b_4 >$ as WCP with a cost of 14:

$$WCET = cm_1 + cm_2 + cm_4 = 14$$

Because the nodes b_1, b_2, and b_4 are on the WCP the edges $(1, 2)$ and $(2, 3)$ are weighted with the maximal execution count, which is in this case 1. The edges from and to b_3 are weighted with 0, since this node is not on the WCP.

So on assigning the basic blocks to the BBS–ISP only the nodes on the WCP are considered. Furthermore, a jump penalty jp of 2 is considered. Under assumption that all 3 basic blocks that are on the WCP fit into the BBS–ISP the benefit in LP (4.15) is optimal if all blocks are assigned to the scratchpad. Since all basic blocks are in the same memory no jump penalties occur. Thus the cost of the WCP is calculated as:

$$cost^*_{WCP} = cs_1 + cs_2 + cs_4 = 7$$

Due to b_3 is now more expensive than b_2 and jump penalties to connect b_3 occur the WCP path changes after the assignment to $< b_1, b_3, b_4 >$. Thus the WCET is as follows:

$$WCET^* = cs_1 + jp_{(1,3)} + cm_3 + jp_{(3,4)} + cs_4 = 15$$

According to the consideration of the jump penalties for the connecting jumps the WCET is impaired by assigning the basic blocks b_1, b_2 and b_4 to the BBS–ISP. Hence, the knapsack approach is not able to ensure that the WCET is improved, it cannot even guarantee that the WCET is not changed to the worse. This is caused by the fact that the knapsack approach considers only the precalculated WCP. ♦

Therefore, the knapsack approach is not recommended, if an application contains multiple paths that can become the WCP. As for the simplified case the scratchpad assignment can be obtained by using the Algorithm 4.1.

WCP-sensitive Basic Block Assignment Without Additional Penalties

For the case that no penalties are caused by relocating a basic block to the BBS–ISP the WCP-sensitive assignment as described by the linear program (4.10) can be applied with the simplification that every basic block i is also a snippet s which can be assigned to the scratchpad. Therefore, the Equation (4.9) can be reduced to:

$$\forall i \in basic\_blocks : \forall (i,j) \in E : cx_{(i,j)} = cm_{(i,j)} \cdot (1 - x_i) + cs_{(i,j)} \cdot x_i$$

Then the LP is as follows:

$$\begin{aligned} &minimize && w_{program\_entry} \\ &subject\ to && \sum_{\forall i \in basic\_blocks} x_i \cdot size_i \leq size_{ISP} \\ &with && x_i = \begin{cases} 0 & if \text{ basic block } i \text{ is located in off-chip memory} \\ 1 & if \text{ basic block } i \text{ is located in scratchpad memory} \end{cases} \end{aligned}$$

Using this linear program in Algorithm 4.2 the optimal basic block assignment for the BBS–ISP can be found, if the jump and size penalties are not considered.

WCP-sensitive Basic Block Assignment Including Jump and Size Penalties

For the WCP-sensitive basic block assignment the penalties caused by additional connective jumps between assigned and unassigned basic blocks need to be considered to find the optimal set of basic blocks that are assigned to the scratchpad. Otherwise it is possible that the obtained set of assigned basic blocks lead to a WCET increase, due to jump penalties, or may not even fit the scratchpad, because of increased basic block sizes.

To take the jump penalties into account the formula for the calculation of the cost of all nodes in a non-cyclic control flow, represented by a DAG, needs to be updated. Therefore, the Equation (4.9) is extended by the jump penalty from the block i to its successor block j:

$$\forall i \in basic\_blocks : \forall (i,j) \in E : cx_{(i,j)} = cm_{(i,j)} \cdot (1 - x_i) + cs_{(i,j)} \cdot x_i + jp_{(i,j)} \cdot (x_i \oplus x_j) \quad (4.19)$$

Due to the encapsulation of the additional jump penalty into the cost of the basic blocks, all other equations from (4.4) to (4.8) do not need to be altered to build the jump-penalty-aware linear program.

The different jump penalties jp are defined as for the knapsack-based basic block assignment as defined by Equation (4.17). By applying the changes introduced by the Equation (4.19) the WCP-sensitive assignment considers the additional cost caused by conserving the control flow when basic blocks are assigned to a scratchpad. To also take the basic block size penalties into account the Equation (4.18) needs to be integrated. Then the linear program for a WCP-sensitive basic block assignment including jump and size penalties is as follows:

$$
\begin{aligned}
&\textit{minimize} \quad w_{program\_entry} \qquad (4.20)\\
&\textit{subject to} \quad \sum_{\forall i \in basic\_blocks} x_i \cdot size_i^p \quad \leq size_{ISP}\\
&\textit{with} \quad x_i = \begin{cases} 0 & if \text{ basic block } i \text{ is located in off-chip memory} \\ 1 & if \text{ basic block } i \text{ is located in scratchpad memory} \end{cases}\\
&\qquad jp_{(i,j)} = \begin{cases} jp_{ca} & if \text{ j is reached from i by continuous addressing} \\ jp_{js} & if \text{ j is reached from i by short jump} \\ jp_{jl} & if \text{ j is reached from i by long jump} \\ jp_{cl} & if \text{ j is reached from i by call} \\ 0 & otherwise \end{cases}\\
&\qquad size_i^p = \; size_i\\
&\qquad + \sum_{\forall (i,j) \in E} \begin{cases} (x_i \wedge \neg x_j) \cdot sp_{ca} & if \text{ j is reached from i by continuous addressing} \\ (x_i \wedge \neg x_j) \cdot sp_{js} & if \text{ j is reached from i by (un)conditional short jump} \\ (x_i \wedge \neg x_j) \cdot sp_{jl} & if \text{ j is reached from i by (un)conditional long jump} \\ (x_i \wedge \neg x_j) \cdot sp_{cl} & if \text{ j is reached from i by call} \\ 0 & otherwise \end{cases}
\end{aligned}
$$

The worst-case execution cost $w_{program\_entry}$ embeds the control flow of the whole application, which is constructed as shown for the generic case in Section 4.1.1. The Algorithm 4.2 can be applied with minimal changes to allow assigning single basic blocks and taking their penalties on assignment into account. The WCP-sensitive basic block assignment that considers the penalties introduced by conserving the control flow of relocated basic blocks delivers the optimal assignment and ensures that the WCET is not impaired by the assignment. Despite of minor changes, e.g. regarding the calculation of the basic block cost, this approach was proposed by Falk and Kleinsorge [2009].

Basic Block Reconnection and Code Relocation

Once the assigned basic blocks are determined, the application needs to be altered to add the correct jump targets and preserve the control flow. This could be done for example in the linked executable by the analysis tool as proposed in Section 5.2. Notice that any changes on the application, e.g. by adding one instruction or replacing a short jump with longer one to reach the basic blocks in the different memory address spaces, can cause a negative impact on the overall application timing due to alignment effects for all subsequent blocks. Therefore, it is essential to keep the alignment of the instructions in the application by code relocation on adding additional or larger instructions. How the basic block assignment is implemented in a WCET analysis tool for a specific processor is presented in Section 5.2.3.

A comparison of the static assignment algorithms is provided in Appendix A. Their evaluation in conjunction with the impact of the D–ISP on the WCET estimate is shown in Section 6.2.

4.2 Analysis of Cache Replacement Policies

The analysis of memories with dynamic content management is much more complex than for memories with fixed content, because the content changes during the execution of the application. To determine the content of dynamic memories (like caches or the D–ISP) a memory analysis has to be performed that takes the execution characteristics of the application and the architectural parameters of the memory, like its size or the applied replacement policy, into account. As caches are the most common dynamic memories there is a wide range of literature on analysis of the cache content available in hard real-time systems research, which is surveyed in Section 2.3.1.

In this section the analysis of different replacement policies for instruction caches is discussed in detail. It is used to explain the analysis of memories to which the D–ISP will be compared in Chapter 6 and lay the foundations for content analysis of the D–ISP described in Section 4.3. Because the described analysis of the cache replacement policies uses abstract interpretation, the basics of abstract interpretation are also introduced in the following. Then the cache content analysis for the different replacement policies (LRU, FIFO, and direct mapped) is described. The shown analysis methods provide a deeper insight in common cache analysis techniques that are based on the work of Ferdinand and Wilhelm [1999] and denote the state of the art in instruction cache analysis.

4.2.1 Memory Analysis Using Abstract Interpretation

According to the definitions in [Nielson et al., 1999, Chapter 4.3], two lattices are introduced L and M. L is the complete lattice, whereas M is a simpler lattice that replaces L. Since an analysis based upon L may be too costly or even incommutable, it will be transferred to the lattice of M. The transfer from L to M is done by an *abstraction function*:

$$\alpha : L \to M$$

Reaching L from M the *concretisation function* is used:

$$\gamma : M \to L$$

This can be written as (L, α, γ, M) forming a Galois connection, if and only if α and γ are monotone functions. According to [Nielson et al., 1999] they have to satisfy:

$$\gamma \circ \alpha \sqsupseteq \lambda l.l \tag{4.21}$$

$$\alpha \circ \gamma \sqsubseteq \lambda m.m \tag{4.22}$$

Meaning that multiple transfers between both lattices will be safe but at the cost of precision.

If an analysis is done in the abstract domain M, a description of the operations performed in the concrete domain L needs to be found for the abstract representation M. Then the analysis can be performed in M with less computational effort. So according to [Nielson et al., 1999]: if $(L_1, \alpha_1, \gamma_1, M_1)$ and $(L_2, \alpha_2, \gamma_2, M_2)$ are Galois connections, then a combined Galois connection can be created:

$$(L_1 \to L_2, \alpha, \gamma, M_1 \to M_2)$$

With definition of α and γ as follows:

$$\alpha(f) = \alpha_2 \circ f \circ \gamma_1$$
$$\gamma(g) = \gamma_2 \circ g \circ \alpha_1$$

Table 4.1: Categorisation of memory accesses in memory analysis

Category	Description
Always Hit (AH)	Every access to a memory reference will be a hit.
Always Miss (AM)	Every access to a memory reference will be a miss.
Not Classified (NC)	The access to a memory reference will be neither always hit nor miss.

Due to monotonicity of $f : L_1 \to L_2$ and $g : M_1 \to M_2$ and the equations (4.21) and (4.22), the traverse between the lattices is safe but with possible loss of precision:

$$\gamma(\alpha(f)) = (\gamma_2 \circ \alpha_2) \circ f \circ (\gamma_1 \circ \alpha_1) \sqsupseteq f$$
$$\alpha(\gamma(g)) = (\alpha_2 \circ \gamma_2) \circ g \circ (\alpha_1 \circ \gamma_1) \sqsubseteq g$$

This relation is depicted in the commutative diagrams:

$$\begin{array}{ccc} L_1 & \xrightarrow{f} & L_1 \\ {\scriptstyle\gamma_1}\uparrow & & \downarrow{\scriptstyle\alpha_2} \\ M_1 & \xrightarrow[\alpha(f)]{} & M_2 \end{array} \qquad \begin{array}{ccc} L_1 & \xrightarrow{\gamma(g)} & L_1 \\ {\scriptstyle\alpha_1}\downarrow & & \uparrow{\scriptstyle\gamma_2} \\ M_1 & \xrightarrow[g]{} & M_2 \end{array} \tag{4.23}$$

Thus if a concrete representation is transferred into an abstract domain on which the analysis is performed, the result transferred back into concrete domain provides a safe approximation of the result of an analysis in concrete domain. So if the transfer between both domains α_1 and γ_2 and the analysis represented by g is less complex and compute intensive than f, the use of the abstract domain is worth. The preciseness of the abstract analysis then will depend on the quality of the used transfer and analysis functions.

The properties and their formal proof of a Galois connection are described in whole in [Nielson et al., 1999] and are the foundations for the cache memory analysis proposed by [Ferdinand et al., 1997; Ferdinand and Wilhelm, 1999]. The lattices L and M will be mapped to concrete and abstract cache states. An abstract cache state domain will be denoted in the following as $\hat{C}$ and the concrete domain will be denoted as C. The cache analysis semantics used for abstract interpretation are presented by [Reineke, 2008] in detail.

4.2.2 LRU Replacement Policy

An analysis for a cache with LRU replacement policy (I–Cache$_{LRU}$) is described originally in [Ferdinand et al., 1997]. It uses the abstract interpretation introduced in the previous section. The cache analysis is split into a *must* and *may* analysis. The must analysis determines which memory reference has to be in the cache at different times during program execution. Thus it is able to categorize memory accesses as *always hit*. The may analysis provides information about memory blocks that might be in the cache. A memory block that is not in an abstract may state can be considered as *always miss*. A memory block that is in the may set but not in the must set cannot be categorized and is marked as *not classified* (NC). In Table 4.1 these memory access categories are shown.

The abstract memory states of the cache are maintained by two functions: `update` and `join`. The `update` function handles the changes to the memory state on access of a memory reference, which results either in hit or miss. The `join` function is needed, if several memory states needs to be merged. This is necessary when different program paths are merged in one control flow, e.g. after an `if-then-else` control flow.

In the following a short description of the cache analysis for fully associative caches with LRU replacement policy is given. The analysis and its formulation strictly follows the work of Ferdinand and Wilhelm [1999] and Reineke [2008] and uses their terminology.

A cache state c of concrete cache domain C maps memory blocks (also denoted as memory references) to one of n cache lines. So a concrete cache state is represented as n-tuple:

$$c = [b_0, b_1, \cdots b_{n-1}]$$

The line of a memory reference can be accessed by $c(r)$.

$$c(r) = \begin{cases} a & if\ c = [b_0, \cdots b_{n-1}], b_a = r \\ \bot & if\ r \notin c \end{cases}$$

If a memory reference is not in the cache state, it is denoted by $\bot$. For each memory reference in the cache its age needs to be known by the analysis. The age is necessary for the LRU replacement policy, because the oldest memory reference (with the age of $n-1$) gets evicted on a cache miss. The definition of the `age` function is shown in Equation (4.24).

$$age_c(r) = \begin{cases} c(r) & if\ c(r) \neq \bot \\ \infty & otherwise \end{cases} \tag{4.24}$$

An age of ∞ denotes that a memory reference is not in the cache. By this definition the line directly corresponds to the age of the memory reference within that line. So the altering of the ages in the cache can be represented by moving references to different cache lines. This is not suitable for a hardware implementation of a cache, but this abstraction simplifies the formulation of the analysis without any influence on the cache behaviour.

An abstract cache state $\hat{c}$ of abstract cache domain $\hat{C}$ may contain multiple memory references within one of n abstract cache lines. The age of the abstract cache line is denoted by the lower index on the closing bracket. So in the example below the $age_{\hat{c}}(r_3) = 1$:

$$\hat{c} = [\{r_0, r_1\}_0, \{r_3\}_1, \{\bot\}_2, \cdots \{\cdots\}_n]$$

One block can only be in one cache line in the abstract cache state, resulting that the age of a memory reference can be determined by the following equations:

$$age_{\hat{c}}(r) = \begin{cases} \hat{c}(r) & if\ \hat{c}(r) \neq \bot \\ \infty & otherwise \end{cases} \tag{4.25}$$

$$\hat{c}(r) = \begin{cases} a & if\ \hat{c} = [\{\cdots\}_0, \cdots, \{\cdots, b_x, \cdots\}_a, \cdots \{\cdots\}_{n-1}], b_x = r \\ \bot & if\ r \notin c \end{cases}$$

Must-Analysis

The must analysis determines what memory reference has to be in the cache at an arbitrary point during program execution. This is done by holding the maximum age of each memory reference of a set of concrete states in the corresponding abstract cache state ($\hat{C}_{must}$). Hence, the must analysis provides information about a sure cache hit.

The abstraction function α^{LRU}_{must} and the concretisation function γ^{LRU}_{must} for the LRU cache must analysis are defined by the equations below.

$$\begin{aligned} \alpha^{LRU}_{must} &: \mathcal{P}(C) \mapsto \hat{C}_{must} \\ \alpha^{LRU}_{must}(C) &= \lambda age_{\hat{c}}(r).\max_{\forall c \in C}(age_c(r)) | \forall r \in R \end{aligned} \tag{4.26}$$

The abstraction function creates from a set of concrete cache states[5] an abstract cache state in domain $\hat{C}_{must}$ by selecting the maximum age of each memory reference (from the set of all memory references R). Notice that memory references that are not in the cache state have an age of ∞. And thus if a memory references is not in all concrete states from the set, the memory reference is not in the abstract cache state, which is denoted by an age of ∞.

$$\begin{aligned} \gamma_{must}^{LRU} &: \hat{C}_{must} \mapsto \mathcal{P}(C) \\ \gamma_{must}^{LRU}(\hat{c}) &= \{c \in (C) | \forall r \in R : age_c(r) \leq age_{\hat{c}}(r)\} \end{aligned} \tag{4.27}$$

The concretisation function builds all sound concrete cache states of C in which the age of each memory reference is not older than the age of this memory references in the abstract cache state. Notice that in a sound concrete cache state one line can only contain one memory reference.

Definition 1. The `update` function for the abstract cache state using the least recently used (LRU) replacement policy creates a new abstract cache state ($\hat{C}_{must}$) by altering a given abstract cache state on access of a memory reference (R):

$$update_{must}^{LRU} : \hat{C}_{must} \times R \mapsto \hat{C}_{must}$$

The `update` function builds the abstract cache state $\hat{c}'$ and is defined as:

$$update_{must}^{LRU}(\hat{c}, r) = \lambda \hat{c}'(u). \begin{cases} 0 & if\ r = u & (1) \\ \hat{c}(u) & if\ \hat{c}(r) \leq \hat{c}(u) & (2) \\ \hat{c}(u) + 1 & if\ \hat{c}(r) > \hat{c}(u) \wedge \hat{c}(u) < n - 1 & (3) \\ \bot & if\ \hat{c}(r) = \bot \wedge \hat{c}(u) = n - 1 & (4) \end{cases} \tag{4.28}$$

The definition distinguishes the four cases:

(1) The reference that is accessed (r) is moved to line 0, which represents the age 0. This is independent of the fact if r was already in the cache state or not.

(2) The memory reference u is not moved for the new abstract state $\hat{c}'$, if the age of the accessed reference r is less or equal.

(3) The memory reference u is moved to the next line in the new abstract state $\hat{c}'$, if the age of the accessed reference r was higher. So because the accessed reference r is moving to the front, all references that are younger than $\hat{c}(r)$ are aged by one.

(4) If the accessed reference r was not in the abstract cache state the oldest entry, with age $n - 1$ is evicted. This is marked by the position $\bot$ that represents the age ∞.

■

Example. An example for the update function is shown in Figure 4.5. There the memory reference x is accessed and moved to the front and by this the age of the reference a is increased. Due to condition (2) in Equation (4.28) y and b are left unchanged. The reference y keeps the age after accessing x, because it represents the abstract cache position for the must analysis. That means the age of x and y in all sound concrete states is at most the age shown in Figure 4.5. So in all concrete cache states that are possible either x is younger than y or not, but both can not be older than their age in the abstract state $\hat{c}$.

[5] $\mathcal{P}(C)$ denotes the power set of C.

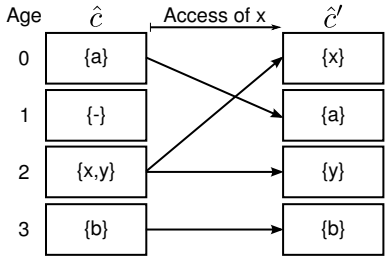


Figure 4.5: Update function for the LRU cache must analysis (derived from [Ferdinand and Wilhelm, 1999])

- Assume a concrete state c in which $age_c(x) < age_c(y)$ and x is accessed, then the age of x is set to 0 but the age of y is not affected by this.
- Assume a concrete state c in which $age_c(x) > age_c(y)$ and x is accessed, then the age of x is set to 0 and the age of y has to be increased, but not beyond the age of x in the initial state.

Notice that no concrete state is possible in which x and y have the same age. Thus the memory reference y keeps its age in the updated abstract state $\hat{c}'$ shown in Figure 4.5 ♦

If several control flows are joined, their abstract cache states have to be joined for the must analysis. Therefore, the `join` function is introduced.

Definition 2. The `join` function combines two abstract cache states to one:

$$join^{LRU}_{must} : \hat{C}_{must} \times \hat{C}_{must} \mapsto \hat{C}_{must}$$

It creates the joined abstract cache state $\hat{c}_3$ is defined as:

$$join^{LRU}_{must}(\hat{c}_1, \hat{c}_2) = \hat{c}_3, \quad where\ \forall r \in \hat{c}_1 \wedge \hat{c}_2 : \hat{c}_3(r) = \max(\hat{c}_1(r), \hat{c}_2(r)) \tag{4.29}$$

The join transfers all memory references that are in both abstract cache states into the new cache state with the maximum age of both entries. Thus the join can be seen as an intersection of states applying the maximum age for the memory references. ■

Example. An example for the join of two abstract states in the must analysis is shown in Figure 4.6. It shows how the intersection with maximum age of both abstract memory states is constructed. ♦

May-Analysis

The `update` and `join` functions for the may analysis of the LRU cache are used to create the abstract cache state that contains all memory references that could be in the cache. It is needed to check the *always miss* property for the memory references. Therefore, the minimum age of a memory reference in every possible concrete cache state is represented by the abstract cache state ($\hat{C}_{may}$), which is used for the may analysis.

The abstraction function α^{LRU}_{may} and the concretisation function γ^{LRU}_{may} for the LRU cache may analysis are defined by the equations below.

$$\begin{aligned} \alpha^{LRU}_{may} &: \mathcal{P}(C) \mapsto \hat{C}_{may} \\ \alpha^{LRU}_{may}(C) &= \lambda age_{\hat{c}}(r).\min_{\forall c \in C}(age_c(r)) | \forall r \in R \end{aligned} \tag{4.30}$$

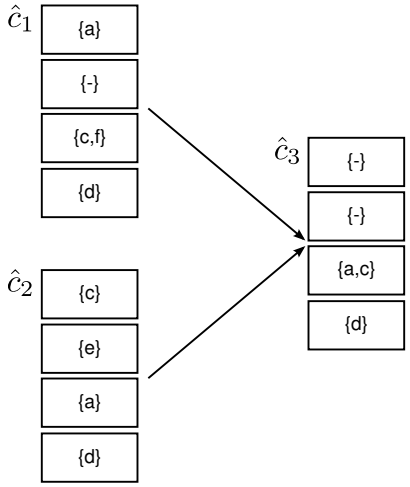


Figure 4.6: Join function for the LRU cache must analysis (derived from [Ferdinand and Wilhelm, 1999])

The abstraction function creates from a set of concrete cache states an abstract cache state in domain $\hat{C}_{may}$ by selecting the minimum age of each memory reference from the concrete cache states.

$$\begin{aligned} \gamma_{may}^{LRU} &: \hat{C}_{may} \mapsto \mathcal{P}(C) \\ \gamma_{may}^{LRU}(\hat{c}) &= \{c \in C | \forall r \in R : age_c(r) \geq age_{\hat{c}}(r) \end{aligned} \tag{4.31}$$

The concretisation function builds all sound concrete cache states of C in which the age of each memory reference is at least as old as the age of this memory references in the abstract cache state $\hat{c}$. Notice that the build concrete states are sound, if and only if there is at most one memory reference assigned to a certain cache line, i.e. each memory reference in c has to have a different age in c ($age_c(r)$).

Definition 3. The `update` function alters a given abstract cache state by accessing a memory reference and so creates a new abstract cache state:

$$update_{may}^{LRU} : \hat{C}_{may} \times R \mapsto \hat{C}_{may}$$

It builds the new abstract cache state $\hat{c}'$ and is defined as:

$$update_{may}^{LRU}(\hat{c}, r) = \lambda \hat{c}'(u). \begin{cases} 0 & if\ r = u & (1) \\ \hat{c}(u) & if\ \hat{c}(r) < \hat{c}(u) & (2) \\ \hat{c}(u) + 1 & if\ \hat{c}(r) \geq \hat{c}(u) \wedge \hat{c}(u) < n - 1 & (3) \\ \bot & if\ \hat{c}(r) = \bot \wedge \hat{c}(u) = n - 1 & (4) \end{cases} \tag{4.32}$$

The definition distinguishes the four cases:

(1) The reference that is accessed (r) is moved to line 0, which represents the age 0. This is independent of the fact if r was already in the cache state or not.

(2) The memory reference u is not moved for the new abstract state $\hat{c}'$, if the accessed reference r is of younger age than u.

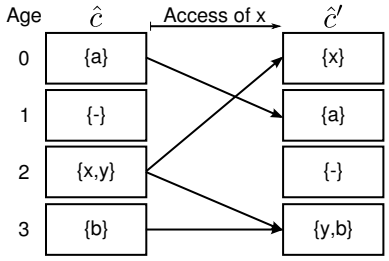


Figure 4.7: Update function for the LRU cache may analysis (derived from [Ferdinand and Wilhelm, 1999])

(3) The memory reference u is moved to the next line in the new abstract state $\hat{c}'$, if the age of the accessed reference r was higher or equal. Because the accessed reference r is moving to the front, all references that were younger or of equal age as r in $\hat{c}$ are aged by one.

(4) If the accessed reference r was not in the abstract cache state the oldest entry, with age $n-1$ is evicted. This is marked by the position $\perp$ that represents the age ∞.

■

Example. The Figure 4.7 shows an example for the update function for the may analysis. The initial abstract state $\hat{c}$ is equal to the example for the must analysis shown before in Figure 4.5. Despite for the memory reference y the behaviour of the update for may analysis is equal. This is caused by the fact that the may analysis holds the best position (with lowest age) of a memory reference of all possible concrete states in the abstract state. To clarify this assume all the concrete states created by Equation (4.31) of the abstract may state for the memory references x and y:

$$\begin{aligned} \hat{c} =& [\{\perp\},\{\perp\},\{x,y\},\{\perp\}] \\ \gamma^{LRU}_{may}(\hat{c}) = c =& \{[\perp,\perp,\perp,\perp], && (1)\\ & [\perp,\perp,x,\perp],[\perp,\perp,x,y],[\perp,\perp,\perp,x], && (2)\\ & [\perp,\perp,y,\perp],[\perp,\perp,\perp,y],[\perp,\perp,y,x]\} && (3) \end{aligned}$$

So the concrete states c of the abstract cache state allow two cases: x is younger than y or not. Notice that the age of x and y in the concrete states can not be younger than the age of them in the abstract state $\hat{c}$.

- Assume a concrete state c in which $age_c(x) < age_c(y)$ and x is accessed, then the age of x is set to 0 but the age of y is not affected by this. Anyhow, the age of y in c is older than $\hat{c}(y)$. This holds for line (2) of the concrete states of $\hat{c}$ shown above.
- Assume a concrete state c in which $age_c(x) > age_c(y)$ and x is accessed, then the age of x is set to 0 and the age of y has to be increased. Since the age of y in the abstract state $\hat{c}$ was already the lowest possible, $age_{\hat{c}'}(y)$ can be safely set to $age_{\hat{c}}(y)+1$. This holds for the concrete states of $\hat{c}$ shown above in line (3).

Line (1) of c does not need to be respected to determine the minimal possible age of y after access of x, because y is not in this concrete cache state – having the age of ∞. Thus the minimal age of y in the abstract cache state $\hat{c}$ has to be increased on access of x as defined in case (3) of Equation (4.32). ♦

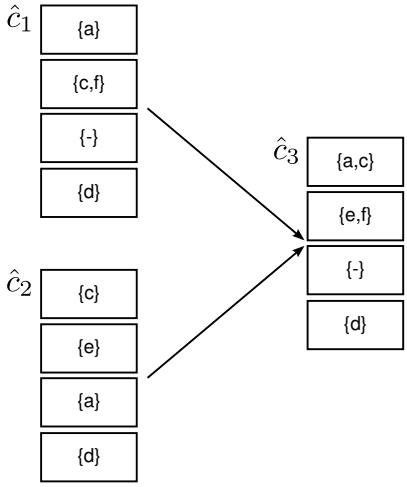


Figure 4.8: Join function for the LRU cache may analysis (derived from [Ferdinand and Wilhelm, 1999])

Definition 4. As for must analysis the `join` function merges two abstract cache states into one abstract cache state:

$$join_{may}^{LRU} : \hat{C}_{may} \times \hat{C}_{may} \mapsto \hat{C}_{may}$$

The `join` creates the new abstract cache state $\hat{c}_3$ and is defined as:

$$join_{may}^{LRU}(\hat{c}_1, \hat{c}_2) = \hat{c}_3, \quad where\ \forall r \in \hat{c}_1 \vee \hat{c}_2 : \hat{c}_3(r) = \min(\hat{c}_1(r), \hat{c}_2(r)) \tag{4.33}$$

By design it adds all memory references that are at least in one of the two input states to the new state. If a memory reference is in both states the minimal age is used for the new state. Thus the join for may analysis can be seen as a union that applies the minimum age for the memory references. ■

Example. The join of two abstract cache states for the may analysis is depicted in Figure 4.8 exemplifying the union of two abstract memory stages using the minimum age of every memory reference. ♦

Using these must and may analyses a proper content analysis for a fully associative cache with LRU replacement policy is possible. The analysis itself is performed in an data flow analysis on the applications control flow graph as discussed in Section 2.6. So every memory access can be classified depending on its presence in the abstract must or may set. The Table 4.2 shows the relation of the abstract cache set content and the classification of the contained memory references.

4.2.3 FIFO Replacement Policy

The FIFO replacement policy (I–Cache$_{FIFO}$) is from its behaviour rather similar to the LRU replacement policy with the exception that accessed elements are not moved to the front of the cache and set to age 0. Unfortunately, this difference has a high impact to the predictability of the FIFO cache, because it is not easy to handle the abstract cache states, if unknown references are accessed (i.e. it is unclear if a memory references is a hit or a miss). This behaviour is described

Table 4.2: Categorisation of memory accesses regarding the abstract must/may sets

Category	**Abstract Must/May Set**
AH	Memory reference is in must set.
AM	Memory reference is not in may set.
NC	Memory reference is in may, but not in must set.

in detail by Reineke and Grund [2008]. It is stated that a k-way set associative FIFO cache with unknown content needs $2k-1$ pairwise different accesses to regain the content information. This is much larger than for LRU, which needs only k accesses. This is due to the fact that a memory reference that was accessed with a cache hit can be evicted on accessing the next most memory reference, which is not the case for LRU.

Furthermore, on insertion of a memory reference in the must set it cannot be simply set to the last-in position. This is only possible, if the access will be definitely a miss, otherwise it has to be inserted at the first-in position (that will be evicted next). To know if an access will be a sure miss a may analysis is needed to determine the position of the insertion of a memory reference into the must set. Nevertheless, a precise may analysis is also hard for the FIFO cache, because the may information of unaccessed memory references does not necessarily age and get evicted. This shows the following example.

Example. Assume the concrete FIFO cache states c and the access to the memory references r and u:

$$\begin{aligned} c = \{c_0, c_1\} &= \{[\bot, u, r], [\bot, u, \bot]\} \\ c|_r = c' = \{c'_0, c'_1\} &= \{[\bot, u, r], [r, \bot, u]\} \\ c'|_u = c'' = \{c''_0, c''_1\} &= \{[\bot, u, r], [r, \bot, u]\} \end{aligned}$$

The leftmost position in the concrete cache state denotes the last-in position (comparable to the age of 0 in an LRU cache) and the rightmost position in the concrete cache state denotes the first-in or first-out position. On a miss a reference is inserted at the last-in position and all references in the cache are shifted by one to the right. The element in the first-in position is evicted.

So in the concrete memory state c'_1, after access of r, r is stored in the last-in position to due to its miss. The position of r in c_0 is not affected by an access of r. Assume that an abstract may state uses the best (leftmost) position of every memory reference in all concrete states as their position. Then the position of r in an abstract may state is the last-in position after access of r from $\hat{c}$. The abstract may position of u is not affected by accessing r and u in c, because a memory reference is never moved if it is already in the concrete cache state.

Hence, to build a proper abstract may analysis for a FIFO cache it is not sufficient to know the best possible position (the leftmost position) of every memory reference of all concrete states. As shown the analysis has to distinguish if a memory reference is in every valid concrete cache state (as for memory reference u) or not (as for reference r). Furthermore, this information is not enough for a correct may analysis, it is also important to consider in which constellation the memory references are in the specific concrete states. This is exemplified in the following.

Assume the following example in which the minimum and the maximum position of both memory references are equal in the concrete states d and e. Furthermore, the two references r and u are not present in every state of concrete states of d and e:

$$
\begin{aligned}
d &= \{[\bot, r, \bot], [\bot, \bot, u], [\bot, u, r]\} \\
d|_r = d' &= \{[\bot, r, \bot], [r, \bot, \bot], [\bot, u, r]\} \\
d'|_v = d'' &= \{[v, \bot, r], [v, r, \bot], [v, \bot, u]\}
\end{aligned}
$$

$$
\begin{aligned}
e &= \{[\bot, r, \bot], [\bot, \bot, u], [\bot, u, \bot], [\bot, \bot, r]\} \\
e|_r = e' &= \{[\bot, r, \bot], [r, \bot, \bot], [r, \bot, u], [\bot, \bot, r]\} \\
e'|_v = e'' &= \{[v, \bot, r], [v, r, \bot], [v, r, \bot], [v, \bot, \bot]\}
\end{aligned}
$$

The minimal and maximal ages of the memory references in the concrete states are the following:

$$
\begin{aligned}
d &: r = [2, \infty]; u = [2, \infty] \\
d' &: r = [1, 3]; u = [2, \infty] \\
d'' &: r = [2, \infty]; u = [3, \infty]; v = [1, 1]
\end{aligned}
$$

$$
\begin{aligned}
e &: r = [2, \infty]; u = [2, \infty] \\
e' &: r = [1, 3]; u = [3, \infty] \\
e'' &: r = [2, \infty]; u = [\infty, \infty]; v = [1, 1]
\end{aligned}
$$

So after accessing r the minimal age of u in d' is different to its age in e'. Resulting in that u will be evicted most likely on the next miss for e'. This happens after access of v: It evicts u from all states in e'', but not from the states in d''. Due to this a correct and precise may analysis for FIFO caches has to take the best and worst position of the memory references in the concrete states and the occurrence of the references in all, some, or none concrete states into account and furthermore also the distribution of the memory references in the concrete states needs to be respected. ⧫

Due to the effects shown in the example above a may analysis of a FIFO cache is complicated. Therefore, Reineke and Grund [2008] propose a may analysis of a k-way FIFO cache by using a $(2k-1)$-way set associative LRU cache. Of course a k-way FIFO cache will have a higher miss rate, but by the *competetivness* of both policies it is guaranteed that a miss in the $(2k-1)$-way LRU cache will also be a miss in the k-way FIFO cache. So the classified definite misses by the may analysis of the larger LRU are safe for the FIFO cache. Using this miss information from the may analysis a more precise must analysis for the FIFO cache is possible, because memory references can be added into the last-in position of the must state on definite miss. This approach is validated for a branch target buffer in [Grund et al., 2009]. Grund and Reineke [2009; 2010a] also propose two additional methods for a preciser FIFO cache analysis. Refer to Section 2.3.1 for a description of these approaches.

Another possibility to analyse the content of a cache using the FIFO policy, due to the lack of safe and simple `update` and `join` functions in the abstract domain, is to keep all concrete states during the analysis. Based on all possible concrete cache states the hit/miss detection can be done without loss of precision, but the usage of all concrete cache states instead of an abstraction domain results in an enormous amount of memory and computational power. This approach is also denoted as *collecting semantics*, refer for example to [Ferdinand and Wilhelm, 1999; Grund and Reineke, 2010a].

4.2.4 Direct Mapped Replacement

A direct mapped cache (I–Cache$_{DM}$) analysis is much easier than the analysis of a set associative cache. This is because the same memory reference will always be located in the same cache line independent of any prior cache access. In the following an analysis of a direct mapped instruction cache is described that is similar to the analysis proposed by Alt et al. [1996], but it uses the same terminology as in Section 4.2.2, which describes the LRU analysis of Ferdinand and Wilhelm [1999].

The function $l(r)$ formulates the relation of mapping the memory references to the cache lines, which is usually done via a modulo function:

$$l(r) = r\%n$$

The number of cache lines available in the cache is n. Thus the cache line of an address in an abstract cache state can be determined by the following equation:

$$\hat{c}(r) = \begin{cases} l(r) & if\ r \in \hat{c} \\ \bot & otherwise \end{cases} \tag{4.34}$$

A memory reference that is not present in the abstract cache state is denoted by $\bot$.

The abstraction and concretisation function to transfer cache states into the abstract respectively concrete domain depend on the type of analysis. Therefore, these functions are described during the must and may analysis later in this section. But due to the fact that the position of a memory reference is fixed in an abstract cache state, the `update` function is equal for the must and may analysis and will be defined as follows.

Definition 5. The `update` function for a direct mapped cache creates a new abstract cache state ($\hat{C}$) by altering a given abstract cache state on access of a memory reference (R).

$$update^{DM} : \hat{C} \times R \mapsto \hat{C}$$

The `update` function builds the abstract cache state $\hat{c}'$ and is defined as:

$$update^{DM}(\hat{c}, r) = \lambda\hat{c}'(u). \begin{cases} l(r) & if\ r = u & (1) \\ \hat{c}(u) & if\ l(u) \neq l(r) & (2) \\ \bot & otherwise & (3) \end{cases} \tag{4.35}$$

The definition distinguishes the three cases:

(1) The reference that is accessed r will be located in the line determined by $l(r)$. This is independent of the fact if r was already in the cache state or not.

(2) Every memory reference that is not conflicting with the cache line in which r is placed, is transferred in the new abstract cache state $\hat{c}'$.

(3) Memory references conflicting with r are evicted.

■

Example. The Figure 4.9 shows an example for the update of one abstract cache state for a direct mapped cache. The memory reference x is accessed, which evicts the memory references c and g that are mapped to the same line from the cache. All other memory references are left unchanged. Notice that in an abstract cache state used by the must analysis there is only one memory reference possible per cache line. Whereas, for the may analysis multiple memory references can be assigned to one abstract cache line. ♦

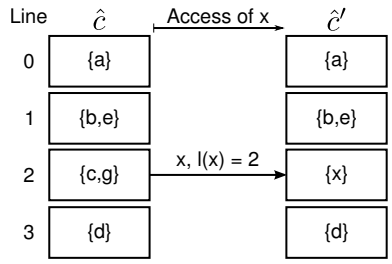


Figure 4.9: Update function for the direct mapped cache analysis (Multiple memory references per cache line are only possible in abstract may states.)

Must-Analysis

The abstraction function α^{DM}_{must} and the concretisation function γ^{DM}_{must} for the direct mapped cache must analysis are defined by the equations below.

$$\begin{aligned} \alpha^{DM}_{must} &: \mathcal{P}(C) \mapsto \hat{C}_{must} \\ \alpha^{DM}_{must}(C) &= \lambda\hat{c}(r).l(r) | \forall r \in R, \forall c \in C : c(r) \neq \bot \end{aligned} \tag{4.36}$$

The abstraction function creates from a set of concrete cache states an abstract cache state in domain $\hat{C}_{must}$ by adding a memory reference into the abstract state, if and only if it is present in every concrete cache state of C. By definition a memory reference r can only be at position $l(r)$. Hence, an abstract must state for the direct mapped cache cannot contain multiple memory references within one cache line.

$$\begin{aligned} \gamma^{DM}_{must} &: \hat{C}_{must} \mapsto C \\ \gamma^{DM}_{must}(\hat{c}) &= \lambda c(r).\hat{c}(r) | \forall r \in R \end{aligned} \tag{4.37}$$

Due to the fact that an abstract cache state for must analysis of the direct mapped cache, cannot contain multiple memory references within one cache line, the concretisation function builds exactly one concrete cache state c that contains the same memory references as $\hat{c}$. Thus, for a sound abstract cache state for must analysis $\hat{c} \in \hat{C}_{must}$:

$$\forall r, s \in \hat{c} \wedge r \neq s : l(r) \neq l(s)$$

always holds.

Example. Assume that an abstract cache state $\hat{c}$ for must analysis contains two references r and s with:

$$l(r) = l(s)$$

There is no possible concrete state c in which both references r and s are contained. So the abstract cache state $\hat{c}$ is not sound for must analysis. ♦

The `join` function for the direct mapped cache depends on the analysis type. For the must analysis the join creates an intersection of both abstract memory states.

Definition 6. The `join` function combines two abstract cache states to one:

$$join^{DM}_{must} : \hat{C} \times \hat{C} \mapsto \hat{C}$$

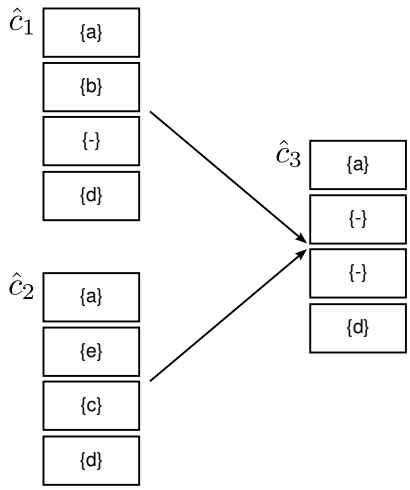


Figure 4.10: Join function for the direct mapped cache must analysis

It creates the abstract cache state $\hat{c}_3$ and is defined as:

$$join^{DM}_{must}(\hat{c}_1, \hat{c}_2) = \hat{c}_3, \quad where\ \forall r \in \hat{c}_1 \wedge \hat{c}_2 : \hat{c}_3(r) = l(r) \tag{4.38}$$

The join transfers all memory references that are in both abstract cache states into the new cache state. The position of the memory references in the cache is not affected by the join, it is fixed by $l(r)$. Therefore, the join function builds an intersection of both states. ■

Example. The Figure 4.10 shows an example of the must analysis joining two abstract memory states for the direct mapped cache to one resulting state. Since only a and d are in both states, only these two memory references are transferred to the joined abstract memory state. ♦

May-Analysis

The abstraction function α^{DM}_{may} and the concretisation function γ^{DM}_{may} for the direct mapped cache may analysis are defined by the equations:

$$\begin{aligned} \alpha^{DM}_{may} &: \mathcal{P}(C) \mapsto \hat{C}_{may} \\ \alpha^{DM}_{may}(C) &= \lambda\hat{c}(r).l(r) | \forall r \in R, \exists c \in C : c(r) \neq \perp \end{aligned} \tag{4.39}$$

The abstraction function creates from a set of concrete cache states an abstract cache state in domain $\hat{C}_{may}$ by adding every memory reference to the build abstract state, if it is at least contained in one of the concrete cache states C.

$$\begin{aligned} \gamma^{DM}_{may} &: \hat{C}_{may} \mapsto \mathcal{P}(C) \\ \gamma^{DM}_{may}(\hat{c}) &= \{c \in C | r \in R : c(r) = \hat{c}(r)\} \end{aligned} \tag{4.40}$$

The concretisation function builds all concrete cache states of C in which a subset of the memory references present in $\hat{c}$ is contained.

In contrast to the must analysis multiple memory references might be located in one cache line of an abstract may cache state. But as for must analysis, an access to an abstract cache line

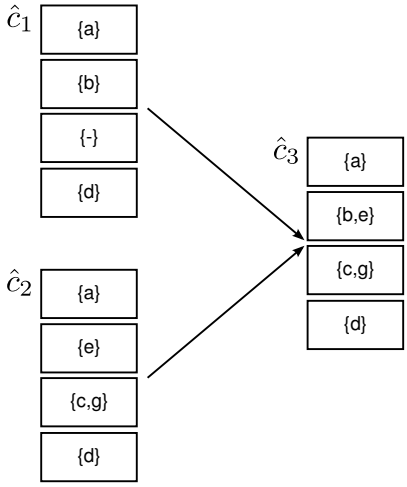


Figure 4.11: Join function for the direct mapped cache may analysis

evicts all other memory references that are maintained in this abstract cache line. This is safe, because on access by a memory reference to a certain cache line all other memory references that could possibly be in the same line as the accessed memory reference are definitely evicted from the cache. So the same update function can be used for must and may analysis.

To get the may information of two abstract cache states that are joined a union of both states can be considered.

Definition 7. The `join` function combines two abstract cache states to one:

$$join_{may}^{DM} : \hat{C} \times \hat{C} \mapsto \hat{C}$$

It creates the abstract cache state $\hat{c}_3$ and is defined as:

$$join_{may}^{DM}(\hat{c}_1, \hat{c}_2) = \hat{c}_3, \quad where\ \forall r \in \hat{c}_1 \vee \hat{c}_2 : \hat{c}_3(r) = l(r) \tag{4.41}$$

The join transfers all memory references that are in at least one abstract cache state into the new cache state. As for must analysis the position of the memory references in the cache is not affected by the join function. ■

Example. A `join` for the may analysis of a direct mapped cache of two abstract cache states is shown in Figure 4.11. As depicted all memory references of both states are transferred to the joined state. ♦

Beside the usage of abstract interpretation other analysis methods for direct mapped caches are proposed, like the work of Arnold et al. [1994] or Li et al. [1995].

4.3 Analysis of D–ISP Replacement Policies

This section presents the content analysis of the D–ISP for the replacement policies proposed in Section 3.2.4. At first the analysis of the D–ISP using the LRU replacement policy is discussed, which is based on the LRU cache analysis using abstract interpretation of Ferdinand and Wilhelm [1999] that was described in the last section. For the D–ISP with FIFO and stack-based

replacement policy no sound abstract representation were found. Therefore, these replacement policies are analysed by maintaining the set of all possible concrete states of the D–ISP during the analysis. This approach is also known as *collecting semantics*, refer for example to [Ferdinand and Wilhelm, 1999; Grund and Reineke, 2010a].

4.3.1 LRU Replacement Policy

The D–ISP$_{LRU}$ analysis is based on the LRU cache analysis by Ferdinand and Wilhelm [1999] that was described in Section 4.2.2. The main difference of the D–ISP analysis and the cache analysis is that the memory references represent functions, which have a non-uniform size. Therefore, the standard LRU cache analysis has to be adapted to support functions instead of cache lines.

A concrete D–ISP state $d \in D$ is represented as a queue in which multiple functions are hold. The concrete D–ISP state is represented by an n-tuple with n as the number of different functions of an arbitrary application that may be contained in the D–ISP:

$$d = [g_0, g_1, \cdots g_n]$$

The positions of the functions in the queue represent the age of the functions. New functions are always added at the front of the queue, thus all other functions are moved towards the end of the queue. By this the age of the functions in the scratchpad is increased. Also if a function is accessed that is already in the concrete D–ISP state, it is moved to the beginning and set to the age of 0. All other function that were in front of the activated function will then be be moved behind the activated one. If a function is moved by the activation of another function, its age is always increased by the size of the activated function. The size of a function is defined by size(f). Functions reserve the amount of memory determined by their size in the concrete D–ISP representation exclusively, i.e. one function cannot overlap with another in any concrete state d. This is is depicted by the equation:

$$\forall f, \forall g \in d \wedge d \in D : f \cap_d g = \begin{cases} \emptyset & if\ f \neq g \\ f & otherwise \end{cases}$$

So the intersection of two different functions in the concrete D–ISP memory state d is always the empty set or the two functions are equal ($f = g$).

Example. Assume the following LRU D–ISP memory state d:

$$\begin{aligned} d &= [g_0, g_1, g_2] \\ d_{|f} &= [f, g_0, g_1, g_2] \\ d_{|f,g_1} &= [g_1, f, g_0, g_2] \end{aligned}$$

In the initial state d the function f is accessed, causing the load of the function f into the concrete D–ISP memory. Due to the LRU replacement policy the function f is added with age 0 and the age of all other functions is increased by the amount of memory that the function f uses, which is the size of the function. In this case it is assumed that the size of the D–ISP is not exceeded on load of function f such that the function g_2, which has the highest age in d has not to be evicted. After loading f the function g_1 is accessed. Thus its age is set to 0. The age of the functions with lower age than g_1 in the initial state $d_{|f}$ will be increased by the size of function g_1. In this example these functions are f and g_0. ♦

Because in the concrete domain a function cannot overlap with another and no gaps are possible between functions, the age of a function can be represented as age_d. The age_d function

determines the beginning position of a function and the end position of a function is determined by the end_d function.

$$age_d(f) = \begin{cases} \sum_{i=0}^{m} \text{size}(g_i) & if\ d = [g_0, \cdots g_m, f, \cdots g_n] \\ \infty & otherwise \end{cases} \tag{4.42}$$

$$end_d(f) = age_d(f) + \text{size}(f) \tag{4.43}$$

An abstract D–ISP state $\hat{d} \in \hat{D}$ holds an arbitrary number of functions. Because it is an abstract representation of multiple concrete states the functions in can completely or partly overlap each other. The abstract D–ISP state is an n-tuple of n different functions of an arbitrary application that may be contained in the D–ISP:

$$\hat{d} = [g_0, g_1, \cdots g_n]$$

The beginning position of a function in the abstract memory state is accessed by $\hat{d}(f)$. Based on this the age of a function can be defined as the position of it in the abstract memory state:

$$age_{\hat{d}}(f) = \begin{cases} \hat{d}(f) & if\ f \in \hat{d} \\ \infty & otherwise \end{cases} \tag{4.44}$$

$$end_{\hat{d}}(f) = age_{\hat{d}}(f) + \text{size}(f) \tag{4.45}$$

In the abstract D–ISP state domain $\hat{D}$ it is possible that two arbitrary functions may intersect in a sound abstract state ($\exists f, g \in \hat{d} \wedge f \neq g : f \cap_{\hat{d}} g \neq \emptyset$). On transfer to concrete domain D overlapping functions need to be resolved by the concretisation function γ^{LRU} to build sound concrete states. The transfer functions between abstract and concrete domains depend on the analysis type and will be introduced during description of the must and may analysis.

Must-Analysis

The must analysis for the D–ISP$_{LRU}$ determines which function has to be in the scratchpad at arbitrary points during execution. The must analysis is needed to predict D–ISP hits for functions on call or return. Abstract D–ISP states ($\hat{D}_{must}$) are used to represent every function in the scratchpad with its maximum age concerning any possible concrete state.

The abstraction function α^{LRU}_{must} and the concretisation function γ^{LRU}_{must} for the LRU cache must analysis are defined by the equations below.

$$\begin{aligned} \alpha^{LRU}_{must} &: \mathcal{P}(D) \mapsto \hat{D}_{must} \\ \alpha^{LRU}_{must}(D) &= \lambda age_{\hat{d}}(f).\max_{\forall d \in D}(age_d(f)) | \forall f \in F \end{aligned} \tag{4.46}$$

The abstraction function creates from a set of concrete D–ISP states an abstract D–ISP state in domain $\hat{D}_{must}$ by selecting the maximum age of each function (from the set of all functions F). Notice that functions that are not in the abstract D–ISP state have an age of ∞.

$$\begin{aligned} \gamma^{LRU}_{must} &: \hat{D}_{must} \mapsto \mathcal{P}(D) \\ \gamma^{LRU}_{must}(\hat{d}) &= \{d \in D | \forall f \in F : age_d(f) \leq age_{\hat{d}}(f)\} \end{aligned} \tag{4.47}$$

The concretisation function builds all sound concrete D–ISP states in D in which the age of each function is at most as old as the age of this function in the abstract D–ISP state $\hat{d}$. Notice that

in contrast to the abstract domain by definition of the concrete D–ISP domain functions cannot overlap each other in any state in D.

As for the concrete domain abstract D–ISP memory states can be updated only on call and return. Therefore, the `update` function needs to be triggered only on function activation.

Definition 8. The `update` function for an abstract D–ISP state using the LRU replacement policy creates a new abstract D–ISP state ($\hat{D}_{must}$) by altering a given abstract D–ISP state on activation of a function (F).

$$update^{LRU}_{must} : \hat{D}_{must} \times F \mapsto \hat{D}_{must}$$

The `update` function builds the new abstract state $\hat{d}'$ and is defined as follows:

$$update^{LRU}_{must}(\hat{d}, f) = \lambda age_{\hat{d}'}(g).\begin{cases} 0 & if\ f = g \quad (1) \\ \sigma^{must}_{\hat{d}}(g, f) & if\ f \bigcap_{\hat{d}} g \neq \emptyset \\ & \wedge\ \sigma^{must}_{\hat{d}}(g, f) + \text{size}(g) \leq size_{D\text{–}ISP} \quad (2) \\ age_{\hat{d}}(g) & if\ f \bigcap_{\hat{d}} g = \emptyset \wedge age_{\hat{d}}(f) < age_{\hat{d}}(g) \quad (3) \\ age_{\hat{d}}(g) + \text{size}(f) & if\ f \bigcap_{\hat{d}} g = \emptyset \wedge age_{\hat{d}}(f) > age_{\hat{d}}(g) \\ & \wedge\ end_{\hat{d}}(g) + \text{size}(f) \leq size_{D\text{–}ISP} \quad (4) \\ \infty & otherwise \quad (5) \end{cases} \tag{4.48}$$

The update function distinguishes the following cases:

(1) The accessed function f is set to the age 0 in $\hat{d}'$, independent if it was already in the input state $\hat{d}$ or not.

(2) If a function g intersects with the activated function f, its age is calculated by $\sigma^{must}_{\hat{d}}(g, f)$, using equation (4.51) that is defined later. An intersection of two functions, denoted by $\bigcap_{\hat{d}}$ occurs, if the start and the end age of both functions have a common interval. This is represented by a non empty set ($\neq \emptyset$). The function g is only transferred to $\hat{d}'$, if g with the updated age does not exceed the scratchpad size ($\sigma^{must}_{\hat{d}}(g, f) + \text{size}(g) \leq size_{D\text{–}ISP}$).

(3) The age of a function g is not altered if its is older than the accessed function f.

(4) For a function g with lower age than the age of the accessed function f in the initial state $\hat{d}$, its age is increased by the size of the function f. If the function f is not in the initial memory state $\hat{d}$ (i.e. $age_{\hat{d}}(f) = \infty$), the age of every function in $\hat{d}$ has to be increased. The function g is not transferred to $\hat{d}'$, if it would exceed the scratchpad size after increasing its age ($end_{\hat{d}}(g) + \text{size}(f) \leq size_{D\text{–}ISP}$).

(5) A function g is evicted, if the size of the scratchpad is exceeded. Because of the indivisibility of functions in the D–ISP, this is detected when the end of a function is beyond the end of the scratchpad.

■

Assume the case (2) from Equation (4.48) in which two functions overlap in the abstract must state. If the abstract state is transferred into concrete domain (by γ^{LRU}_{must}), the functions cannot overlap, because this is impossible in the scratchpad memory. Then the maximum age of a function that overlaps with the activated one in abstract domain has to be determined by

inspecting the worst possible memory states in the concrete domain D. Therefore, this case is discussed in detail.

Assume an abstract memory state for must analysis $\hat{d}$ in which the functions f and g overlap ($f \bigcap_{\hat{d}} g \neq \emptyset$) and the minimal age of f is younger than g in the abstract state ($age_{\hat{d}}(f) < age_{\hat{d}}(g)$). Then the concretisation of the abstract memory state delivers two special concrete states: first the maximal possible age of function g but with f older than g, called state $d_{(g)}$:

$$d_{(g)}: \quad \max(age_d(g))|_{age_d(g)<age_d(f)} \tag{4.49}$$

The other state is the similar but for function f, the state $d_{(f)}$:

$$d_{(f)}: \quad \max(age_d(f))|_{age_d(f)<age_d(g)} \tag{4.50}$$

Notice that in state $d_{(g)}$ the maximum possible age of f of the abstract memory state has to be used, otherwise the age of g would not be maximum. Because both functions intersect in the abstract state $\hat{d}$, they are directly concatenated:

$$age_{d_{(g)}}(g) + \text{size}(g) = end_{d_{(g)}}(g) = age_{d_{(g)}}(f)$$

By maximizing the age of g in $d_{(g)}$ according to the definition in (4.49), the maximum age of f from $\hat{d}$ determines the end of g:

$$age_{d_{(g)}}(g) + \text{size}(g) = age_{\hat{d}}(f)$$

So the following equation holds:

$$age_{d_{(g)}}(f) = age_{\hat{d}}(f)$$

This relation is also valid for function g in state $d_{(f)}$:

$$\begin{aligned} age_{d_{(f)}}(f) + \text{size}(f) = end_{d_{(f)}}(f) &= age_{d_{(f)}}(g) \\ age_{d_{(f)}}(f) + \text{size}(f) &= age_{\hat{d}}(g) \\ age_{d_{(f)}}(g) &= age_{\hat{d}}(g) \end{aligned}$$

According to their definition the two concrete states $d_{(g)}$ and $d_{(f)}$ are depicted in the upper part of Figure 4.12. The state $d_{(g)}$ denotes the worst possible concrete state of $\hat{d}$ for function g on later activation of f, because on activation of f function g may be aged, since it is younger than f. In the figure the age of g in $d_{(g)}$ is denoted by the size of the gap x. Then on activation of f it is possible that the age of g in the concrete state may exceed the maximum age of g in $\hat{d}$ (denoted by the vertical line in the figure). So its maximum age in the updated abstract state $\hat{d}'$ may be increased by activation of f in $\hat{d}$. This is depicted in state $d'_{(g)}$, that represents state $d_{(g)}$ after activation of f. The size of the gap x can be computed by:

$$\begin{aligned} \text{size}(x) &= age_{d_{(g)}}(g) = age_{d_{(g)}}(f) - \text{size}(g) \\ &= age_{\hat{d}}(f) - \text{size}(g) \end{aligned}$$

Notice that $age_{\hat{d}}(f) \geq \text{size}(g)$ always holds, otherwise the state $d_{(g)}$ would not be valid and the abstract state $\hat{d}$ cannot be sound. This is because g has to be in the scratchpad for all concrete states (otherwise its age in $\hat{d}$ would be ∞) and it cannot be older than f in the concrete state, if for f the maximal age is assumed (otherwise $age_{\hat{d}}(f) + \text{size}(f) < age_{\hat{d}}(g) \rightarrow f \bigcap_{\hat{d}} g = \emptyset$).

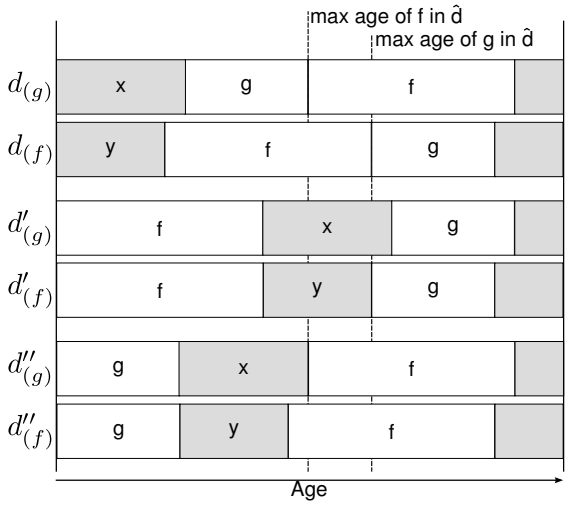


Figure 4.12: Concretisation of an abstract LRU D–ISP must state with intersecting functions f and g

After activation of f in $d_{(g)}$ the age of function g in the resulting state $d'_{(g)}$ is:

$$\begin{aligned} age_{d'_{(g)}}(g) =& \text{size}(f) + \text{size}(x) \\ =& \text{size}(f) + age_{\hat{d}}(f) - \text{size}(g) \end{aligned}$$

For state $d'_{(f)}$, in which g has the maximal age of $\hat{d}$ the following equation holds:

$$age_{d'_{(f)}}(g) = age_{d_{(f)}}(g) = age_{\hat{d}}(g)$$

Due to $age_{d'_{(g)}}(g)$ may be larger than $age_{\hat{d}}(g)$, the new age of g in the abstract state $\hat{d}'$ after activation of f is:

$$\begin{aligned} age_{\hat{d}'}(g) =& \max\left(age_{d'_{(f)}}(g), age_{d'_{(g)}}(g)\right) \\ =& \max\left(age_{\hat{d}}(g), \text{size}(f) + age_{\hat{d}}(f) - \text{size}(g)\right) \end{aligned}$$

In the case shown in Figure 4.12 the age of g in the new abstract state $\hat{d}'$ has to be increased, because:

$$age_{\hat{d}}(g) < \text{size}(f) + age_{\hat{d}}(f) - \text{size}(g)$$

The formula to calculate of the maximum age of g in $\hat{d}'$ also holds for the function f on activation of g in state $\hat{d}$. The resulting abstract state is $\hat{d}''$. The state $d_{(f)}$ and the state $d''_{(f)}$ after activation of g is depicted in Figure 4.12. So the age of f in $\hat{d}''$ can be calculated as follows:

$$\begin{aligned} age_{\hat{d}''}(f) =& \max\left(age_{d''_{(g)}}(f), age_{d''_{(f)}}(f)\right) \\ =& \max\left(age_{\hat{d}}(f), \text{size}(g) + age_{\hat{d}}(g) - \text{size}(f)\right) \end{aligned}$$

The equation above holds, because:

$$\begin{aligned} age_{d''_{(g)}}(f) =& age_{d_{(g)}}(f) \\ =& age_{\hat{d}}(f) \end{aligned}$$

For the case depicted in Figure 4.12 the age of function f in the abstract state $\hat{d}$ is not increased by the activation of g in the concrete state $d_{(f)}$, because:

$$age_{\hat{d}}(f) > \text{size}(g) + age_{\hat{d}}(g) - \text{size}(f)$$

This derivation shows how to obtain the maximum age of the functions f and g, if both functions intersect and one of them is activated. The restriction that f is younger than g can be generalised simply by swapping f and g, because the age of g on activation of f and the age of f on activation of g can be safely calculated for $age_{\hat{d}}(f) < age_{\hat{d}}(g)$. Thus this derivation holds for the general case. Hence, the function $\sigma^{must}_{\hat{d}}(g, f)$ can be defined to calculate the maximum possible age on intersection of two functions in the abstract must domain on update, as needed by case (2) of Equation (4.48).

Definition 9. The $\sigma^{must}_{\hat{d}}(g, f)$ calculates the maximal age of the function g on activation of function f, if and only if both function intersect in the abstract must state $\hat{d}$:

$$\sigma^{must}_{\hat{d}}(g, f) = \max\left(age_{\hat{d}}(g), age_{\hat{d}}(f) + \text{size}(f) - \text{size}(g)\right) \tag{4.51}$$

The result of the function is independent of the age of both functions: g may be younger, older, or of equal age as function f. The only requirement for this function is that g and f intersect in the abstract state $\hat{d}$ ($f \bigcap_{\hat{d}} g \neq \emptyset$). ■

To clarify the maximum age calculation by the `update` function for the D–ISP$_{LRU}$ an example is given below.

Example. Assume the following abstract D–ISP state $\hat{d}$ for must analysis:

$$\hat{d} = [f_{[4,7]}, g_{[4,6]}]$$

The lower index of the functions represent their maximum age and their end position. The complete set of concrete states d created by the concretisation of $\hat{d}$ for the must analysis by γ^{LRU}_{must} (defined in Equation (4.47)) is shown below:

$$\begin{aligned} d = \{&[f_{[0,3]}, g_{[3,5]}], [f_{[0,3]}, x^1_{[3,4]}, g_{[4,6]}], \underbrace{[x^2_{[0,1]}, f_{[1,4]}, g_{[4,6]}]}_{\text{state } d_{(f)}}, \\ &[g_{[0,2]}, f_{[2,5]}], [g_{[0,2]}, x^3_{[2,3]}, f_{[3,6]}], [x^4_{[0,1]}, g_{[1,3]}, f_{[3,6]}], \\ &[g_{[0,2]}, x^5_{[2,4]}, f_{[4,7]}], [x^6_{[0,1]}, g_{[1,3]}, x^7_{[3,4]}, f_{[4,7]}], \underbrace{[x^8_{[0,2]}, g_{[2,4]}, f_{[4,7]}]}_{\text{state } d_{(g)}}\} \end{aligned}$$

To fill the gaps between the functions in the concrete states placeholders x^i are used. On activation of function f it is moved to age 0 and the following set of concrete states $d_{|f}$ is created:

$$\begin{aligned} d_{|f} = \{&[f_{[0,3]}, g_{[3,5]}], [f_{[0,3]}, x^1_{[3,4]}, g_{[4,6]}], [f_{[0,3]}, x^2_{[3,4]}, g_{[4,6]}], \\ &[f_{[0,3]}, g_{[3,5]}], [f_{[0,3]}, g_{[3,5]}, x^3_{[5,6]}], [f_{[0,3]}, x^4_{[3,4]}, g_{[4,6]}], \\ &[f_{[0,3]}, g_{[3,5]}, x^5_{[5,7]}], [f_{[0,3]}, x^6_{[3,4]}, g_{[4,6]}, x^7_{[6,7]}], [f_{[0,3]}, x^8_{[3,5]}, \mathbf{g_{[5,7]}}]\} \end{aligned}$$

The maximum age of function g is calculated by $\sigma^{must}_{\hat{d}}(g, f)$, because both functions intersect in the abstract state $\hat{d}$:

$$\begin{aligned} \sigma^{must}_{\hat{d}}(g, f) &= \max\left(age_{d_{(f)|f}}(g), age_{d_{(g)|f}}(g)\right) \\ &= \max\left(age_{\hat{d}}(g), age_{\hat{d}}(f) + \text{size}(f) - \text{size}(g)\right) \\ &= \max\left(4, 4 + 3 - 2\right) = 5 \end{aligned}$$

If instead of function f function g is activated, the following set of concrete states $d_{|g}$ is generated by updating d:

$$\begin{aligned} d_{|g} = \{&[g_{[0,2]}, f_{[2,5]}], [g_{[0,2]}, f_{[2,5]}, x^1_{[5,6]}], [g_{[0,2]}, x^2_{[2,3]}, f_{[3,6]}], \\ &[g_{[0,2]}, f_{[2,5]}], [g_{[0,2]}, x^3_{[2,3]}, f_{[3,6]}], [g_{[0,2]}, x^4_{[2,3]}, f_{[3,6]}], \\ &[g_{[0,2]}, x^5_{[2,4]}, \mathbf{f}_{[\mathbf{4},\mathbf{7}]}], [g_{[0,2]}, x^6_{[2,3]}, x^7_{[3,4]}, \mathbf{f}_{[\mathbf{4},\mathbf{7}]}], [g_{[0,2]}, x^8_{[2,4]}, \mathbf{f}_{[\mathbf{4},\mathbf{7}]}]\} \end{aligned}$$

For activation of function g the maximum age of f in $\hat{d}_{|g}$ is not increased, because the maximum age of f in $\hat{d}$ is larger than the age of f in state $d_{(f)}$ after activation of g:

$$\begin{aligned} \sigma^{must}_{\hat{d}}(f,g) &= \max\left(age_{d_{(g)|g}}(f), age_{d_{(f)|g}}(f)\right) \\ &= \max\left(age_{\hat{d}}(f), age_{\hat{d}}(g) + \text{size}(g) - \text{size}(f)\right) \\ &= \max\left(4, 4 + 2 - 3\right) = 4 \end{aligned}$$

So the abstract states after activation of f and g with respect to the definition of the `update` function by Equation (4.48) are:

$$\begin{aligned} \hat{d}_{|f} &= [f_{[0,3]}, g_{[5,7]}] \\ \hat{d}_{|g} &= [g_{[0,2]}, f_{[4,7]}] \end{aligned}$$

They can be also build from set of concrete states $d_{|f}$ and $d_{|g}$ using the abstraction function α^{LRU}_{must} defined in (4.46). ♦

On merging control flows, e.g. when branched control flows reunion, also the abstract memory states have to be merged. Joining two abstract must states for D–ISP$_{LRU}$ a similar `join` function as for the LRU cache can be used.

Definition 10. The `join` function combines two abstract D–ISP states to one:

$$join^{LRU}_{must} : \hat{D}_{must} \times \hat{D}_{must} \mapsto \hat{D}_{must}$$

It creates the joined abstract D–ISP state $\hat{d}_3$ and is defined as:

$$join^{LRU}_{must}(\hat{d}_1, \hat{d}_2) = \hat{d}_3, \quad where\ \forall f \in \hat{d}_1 \wedge \hat{d}_2 : age_{\hat{d}_3}(f) = \max\left(age_{\hat{d}_1}(f), age_{\hat{d}_2}(f)\right) \tag{4.52}$$

The join transfers all functions that are in both abstract D–ISP states into the new abstract D–ISP state with the maximum age of both entries. Thus, the join can be seen as an intersection of the memory states applying the maximum age to the functions. ■

Example. The Figure 4.13 shows a join of two arbitrary D–ISP states for must analysis. Notice that during the analysis a join of two states is only possible when the same function is active, since a control flow can be merged at the same stack level only. Due to the LRU replacement policy the active function, which is function f in Figure 4.13, will always be at age 0. All other functions that are also in both states are transferred to $\hat{d}_3$ with their maximum age. ♦

May-Analysis

The may analysis determines the set of functions that might be in the scratchpad at an arbitrary position in the program execution. Therefore, the abstract D–ISP states ($\hat{D}_{may}$) represent all

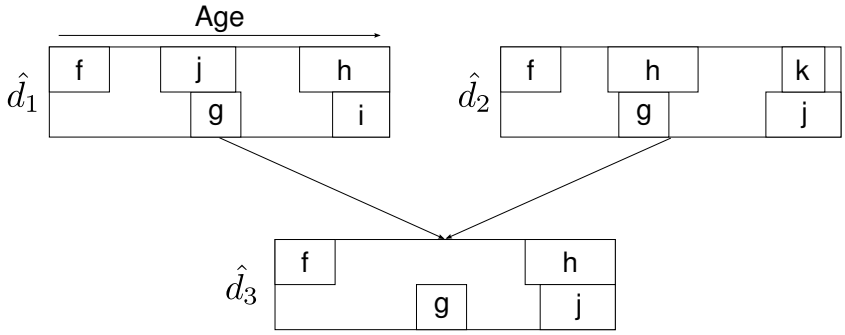


Figure 4.13: Join function for the LRU D–ISP must analysis with f as active function

functions in the scratchpad with their minimum age of all possible concrete states. The abstraction function α^{LRU}_{may} and the concretisation function γ^{LRU}_{may} for the LRU D–ISP may analysis are defined by the equations below.

$$\begin{aligned} \alpha^{LRU}_{may} &: \mathcal{P}(D) \mapsto \hat{D}_{may} \\ \alpha^{LRU}_{may}(D) &= \lambda age_{\hat{d}}(f).\min_{\forall d \in D}(age_d(f)) | \forall f \in F \end{aligned} \tag{4.53}$$

The abstraction function creates from a set of concrete D–ISP states an abstract D–ISP state in domain $\hat{D}_{may}$ by selecting the minimum age of each function of domain F.

$$\begin{aligned} \gamma^{LRU}_{may} &: \hat{D}_{may} \mapsto \mathcal{P}(D) \\ \gamma^{LRU}_{may}(\hat{d}) &= \{d \in D | \forall f \in F : age_d(f) \geq age_{\hat{d}}(f)\} \end{aligned} \tag{4.54}$$

The concretisation function builds all concrete D–ISP states of D in which the age of each function is at least as old as the age of this function in the abstract D–ISP state $\hat{d}$.

On call and return the corresponding abstract D–ISP state has to be altered by the `update` function, that is defined below.

Definition 11. The `update` function for the abstract D–ISP state using the LRU replacement policy creates a new abstract D–ISP state ($\hat{D}_{may}$) by altering a given abstract D–ISP state on activation of a function (F).

$$update^{LRU}_{may} : \hat{D}_{may} \times F \mapsto \hat{D}_{may}$$

The `update` function for may analysis creates the new state $\hat{d}'$ and is defined as follows:

$$update^{LRU}_{may}(\hat{d}, f) = \lambda age_{\hat{d}'}(g). \begin{cases} 0 & if\ f = g & (1) \\ \sigma^{may}_{\hat{d}}(g, f) & if\ f \bigcap_{\hat{d}} g \neq \emptyset & \\ & \wedge\ \sigma^{may}_{\hat{d}}(g, f) + \text{size}(g) \leq size_{D\text{–}ISP} & (2) \\ age_{\hat{d}}(g) & if\ f \bigcap_{\hat{d}} g = \emptyset \wedge age_{\hat{d}}(f) < age_{\hat{d}}(g) & (3) \\ age_{\hat{d}}(g) + \text{size}(f) & if\ f \bigcap_{\hat{d}} g = \emptyset \wedge age_{\hat{d}}(f) > age_{\hat{d}}(g) & \\ & \wedge\ end_{\hat{d}}(g) + \text{size}(f) \leq size_{D\text{–}ISP} & (4) \\ \infty & otherwise & (5) \end{cases} \tag{4.55}$$

The following five cases have to be distinguished:

(1) The accessed function f is always set to age 0 in $\hat{d}'$. This is independent of if the access of f is a hit or miss in $\hat{d}$

(2) On intersection of an arbitrary function g with the activated function f the minimal age of function g is increased to the age determined by $\sigma^{may}_{\hat{d}}(g, f)$ which will be defined by Equation (4.58) later in this section. The function g is only transferred to $\hat{d}'$, if g with the updated age does not exceed the scratchpad size ($\sigma^{may}_{\hat{d}}(g, f) + \text{size}(g) \leq size_{D\text{–}ISP}$).

(3) The age of a function g in the initial state $\hat{d}$ is not altered in $\hat{d}'$ by the access of f, if its age is older than the age of f in $\hat{d}$.

(4) The age of a function g is increased by the size of the activated function f, if its age is less than the age of the activated function in $\hat{d}$. Also g is only transferred to $\hat{d}'$ if the size of the scratchpad is not exceeded by the ageing of g ($end_{\hat{d}}(g) + \text{size}(f) \leq size_{D\text{–}ISP}$).

(5) A function is evicted, if its end position is increased by ageing beyond the end of the scratchpad. This is because of the indivisibility of functions in the D–ISP that does not allow to keep parts of a function in the scratchpad.

■

Assume the case (2) from Equation (4.55) in which two functions overlap in the abstract may state and one of them is activated. Then the new minimal age of the other function in the updated abstract state depends on the age and size of both functions. In the following this case is examined.

Assume an abstract D–ISP state for may analysis $\hat{d}$ in which the function f and g overlap ($f \bigcap_{\hat{d}} g \neq \emptyset$) and f is younger than g ($age_{\hat{d}}(f) < age_{\hat{d}}(g)$). Then the concretisation γ^{LRU}_{may} delivers two specific states that are necessary to determine the new minimum age of g after activation of f: first the state in which g is of minimum age but f is still younger than g:

$$d_{(g1)} : \min(age_d(g))|_{age_d(f)<age_d(g)} \tag{4.56}$$

and second the state in which g has its minimum age and the age of f is not considered:

$$d_{(g2)} : \min(age_d(g)) \tag{4.57}$$

By definition the age of f in the concrete state $d_{(g2)}$ has to be older than the age of g, otherwise both functions would not intersect in the abstract memory state. Because if the function f would be younger than the minimal age of function g in a concrete state, both function cannot intersect in the abstract state:

$$\begin{aligned} if \quad & age_{d_{(g2)}}(f) + \text{size}(f) < age_{d_{(g2)}}(g) \qquad | \; with \; age_{d_{(g2)}}(g) = age_{\hat{d}}(g) \\ then \quad & age_{\hat{d}}(f) \leq age_{d_{(g2)}}(f) \rightarrow age_{\hat{d}}(f) + \text{size}(f) \leq age_{\hat{d}}(g) \rightarrow f \cap_{\hat{d}} g = \emptyset \end{aligned}$$

The age of function g is calculated for both states $d_{(g1)}$ and $d_{(g2)}$ as follows:

$$\begin{aligned} age_{d_{(g1)}}(f) =& age_{\hat{d}}(f) \\ age_{d_{(g1)}}(g) =& age_{d_{(g1)}}(f) + \text{size}(f) = age_{\hat{d}}(f) + \text{size}(f) = end_{\hat{d}}(f) \\ age_{d_{(g2)}}(g) =& age_{\hat{d}}(g) \end{aligned}$$

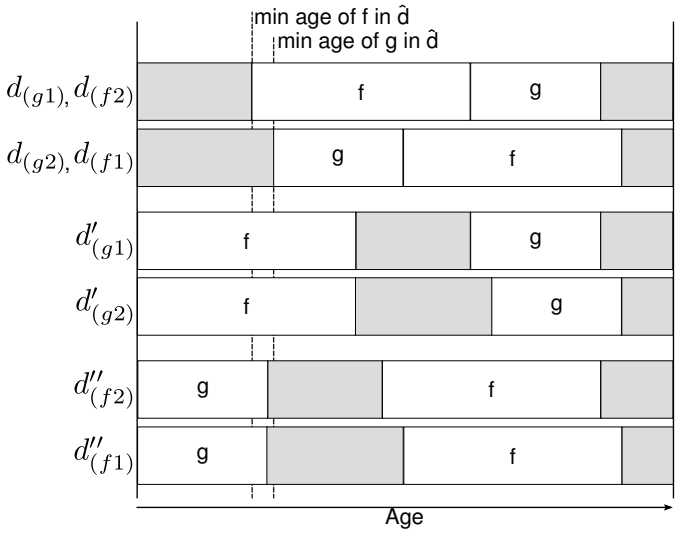


Figure 4.14: Concretisation of an abstract LRU D–ISP may state with intersecting functions f and g

For state $d_{(g1)}$ the minimum end position of f is used as age of g (which is $end_{\hat{d}}(f)$) and for state $d_{(g2)}$ the minimal age of g (which is $age_{\hat{d}}(g)$) determines its age in this concrete state. The Figure 4.14 shows the two specific states $d_{(g1)}$ and $d_{(g2)}$. For state $d_{(g2)}$ the function f is positioned right after function g (which is only of interest later on activation of g). This is possible, because the position of f does not matter for the activation of f, because it is older and will be moved in front of g on activation anyway. Thus f might be also depicted with a higher age or even as not present in the scratchpad.

On activation of f, which builds the abstract state $\hat{d}'$, the minimal age of g in $\hat{d}'$ is either the age derived from $d'_{(g1)}$ or $d'_{(g2)}$:

$$\begin{aligned} age_{d'_{(g1)}}(g) &= age_{d_{(g1)}}(g) = age_{\hat{d}}(f) + \text{size}(f) = end_{\hat{d}}(f) \\ age_{d'_{(g2)}}(g) &= age_{\hat{d}}(g) + \text{size}(f) \\ age_{\hat{d}'}(g) &= \min\left(age_{d'_{(g1)}}(g),\ age_{d'_{(g2)}}(g)\right) \\ &= \min\left(age_{\hat{d}}(f) + \text{size}(f),\ age_{\hat{d}}(g) + \text{size}(f)\right) \end{aligned}$$

For state $d'_{(g1)}$ the age of g is not affected, because f is already younger than g, thus the age of g is not changed. In state $d'_{(g2)}$ the function f was loaded in the scratchpad or moved from an older position than g to age 0. So the function g is aged by the size of f. The minimal value of both ages results in the minimal age for g of all possible concrete states of d', because

- the state $d_{(g1)}$ represents the minimal age of g with f at a younger age, resulting that on activation of f the age of g is not affected,
- and the state $d_{(g2)}$ in which the minimal age of g is used, that is aged by the size of f on its activation.

The Figure 4.14 shows in the third and fourth row the resulting states $d'_{(g1)}$ and $d'_{(g2)}$ after activation of function f. It shows that in this example the minimal age of $\hat{d}'$ is determined by the concrete state $d'_{(g1)}$.

If function g is activated in $\hat{d}$ instead of f its minimal age can be determined correspondingly by minimizing the age of these two specific states for function f. In the Figure 4.14 these states are shown as $d''_{(f1)}$ respectively $d''_{(f2)}$. In contrast to prior discussion the same states in the figure have a different semantic, if taking f into account on activation of function g: The concrete state $d_{(f1)}$ represents the state with minimal age of f but a larger age than function g (with $\min(age_d(f))|_{age_d(g)<age_d(f)}$), and the concrete state $d_{(f2)}$ is the concrete state with minimal age of f and function g is older. So for activation of g in $\hat{d}$ the minimal age of f can be obtained by:

$$age_{\hat{d}''}(f) = \min\left(age_{\hat{d}}(g) + \text{size}(g),\ age_{\hat{d}}(f) + \text{size}(g)\right)$$

In the case shown in the last two rows of Figure 4.14 the minimal age of f in $\hat{d}''$ is given by $d''_{(f2)}$.

By this derivation $\sigma^{may}_{\hat{d}}$ can be defined to calculate the minimal age of two intersecting functions in an abstract may state on activation of one of both. The restriction made for this derivation that f is younger than g in the abstract state ($age_{\hat{d}}(f) < age_{\hat{d}}(g)$) can be relaxed by swapping the functions f and g, because it was shown how to calculate the minimal age of both functions: for g after activation of f and for f after activation of g.

Definition 12. The $\sigma^{may}_{\hat{d}}(g, f)$ calculates the minimal age of the function g on activation of function f, if and only if both functions intersect in the abstract may state $\hat{d}$:

$$\sigma^{may}_{\hat{d}}(g, f) = \min\left(age_{\hat{d}}(f) + \text{size}(f), age_{\hat{d}}(g) + \text{size}(f)\right) \tag{4.58}$$

The calculated age for g is valid for all possible cases regarding the relation of f and g: if g is younger, older, or even of equal age as f. The requirement for $\sigma^{may}_{\hat{d}}(g, f)$ is that both functions intersect in the abstract memory state: $f \cap_{\hat{d}} g \neq \emptyset$. Notice that the calculated minimal age of g is always increased, but not more than the size of f. ■

Now the `update` function as shown in Equation (4.55) is completely defined for the may analysis. To show especially the minimal age calculation for intersecting function an example is given in the following. It describes how the age of functions is calculated in the may analysis on update, if the activated function overlaps with another function.

Example. Assume the following abstract D–ISP state for may analysis:

$$\hat{d} = [f_{[1,4]}, g_{[2,4]}]$$

With a scratchpad size of $size_{D\text{–}ISP} = 7$ the concretisation of $\hat{d}$ using γ^{LRU}_{may} defined in (4.54) results in the set of concrete states d shown below:

$$\begin{aligned} d = \{&[\bot], \\ &[f_{[4,7]}], [f_{[3,6]}], [f_{[2,5]}], \underbrace{[f_{[1,4]}]}_{\text{state } d_{(f2)}}, \\ &[g_{[5,7]}], [g_{[4,6]}], [g_{[3,5]}], \underbrace{[g_{[2,4]}]}_{\text{state } d_{(g2)}}, \\ &\underbrace{[g_{[2,4]}, f_{[4,7]}]}_{\text{state } d_{(f1)}}, \\ &[f_{[2,5]}, g_{[5,7]}], [f_{[1,4]}, g_{[5,7]}], \\ &\underbrace{[f_{[1,4]}, g_{[4,6]}]}_{\text{state } d_{(g1)}}\} \end{aligned}$$

Notice that the concrete states cannot contain any other functions than f and g, because otherwise these other functions would have a minimal age in the abstract D–ISP state. Two functions were selected only for simplicity of the example. Having more functions in the abstract may state the set of concrete states shown above would be a subset of the set of concrete states that are created then.

Two concrete states are highlighted for each function: $d_{(g1)}$ and $d_{(g2)}$ respectively $d_{(f1)}$ and $d_{(f2)}$. From these four marked states the lowest age of function f and g in $\hat{d}$ after activation of one of each functions can be derived. So these states are used by $\sigma^{may}_{\hat{d}}$ defined in Equation (4.58) to calculate the minimal age on function intersection.

After activation of function f the concrete states $d_{|f}$ are altered as follows:

$$\begin{aligned} d_{|f} = \{&[f_{[0,3]}], \\ &[f_{[0,3]}], [f_{[0,3]}], [f_{[0,3]}], [f_{[0,3]}], \\ &[f_{[0,3]}, \underbrace{g_{[8,10]}}_{\text{evicted}}], [f_{[0,3]}, \underbrace{g_{[7,9]}}_{\text{evicted}}], [f_{[0,3]}, \underbrace{g_{[6,8]}}_{\text{evicted}}], \underbrace{[f_{[0,3]}, g_{[5,7]}]}_{\text{state } d_{(g2)|f}}, \\ &[f_{[0,3]}, g_{[5,7]}], \\ &[f_{[0,3]}, g_{[5,7]}], [f_{[0,3]}, g_{[5,7]}], \\ &\underbrace{[f_{[0,3]}, \mathbf{g_{[4,6]}}]}_{\text{state } d_{(g1)|f}}\} \end{aligned}$$

In some states function g is evicted, but its minimal age is defined by the age in $d_{(g1)|f}$. So the minimal age of g after activation of f is:

$$\begin{aligned} age_{\hat{d}_{|f}}(g) &= \sigma^{may}_{\hat{d}}(g, f) = \min(age_{d_{(g1)|f}}(g), age_{d_{(g2)|f}}(g)) \\ &= \min(age_{\hat{d}}(f) + \text{size}(f), age_{\hat{d}}(g) + \text{size}(f)) = \min(4, 5) \\ &= 4 \end{aligned}$$

Assume instead the activation of f the function g is activated. Then the following set of concrete states $d_{|g}$ is created by updating the abstract may state:

$$\begin{aligned} d_{|g} = \{&[g_{[0,2]}], \\ &[g_{[0,2]}, \underbrace{f_{[6,9]}}_{\text{evicted}}], [g_{[0,2]}, \underbrace{f_{[5,8]}}_{\text{evicted}}], [g_{[0,2]}, f_{[4,7]}], \underbrace{[g_{[0,2]}, \mathbf{f_{[3,6]}}]}_{\text{state } d_{(f2)|g}}, \\ &[g_{[0,2]}], [g_{[0,2]}], [g_{[0,2]}], [g_{[0,2]}], \\ &\underbrace{[g_{[0,2]}, f_{[4,7]}]}_{\text{state } d_{(f1)|g}}, \\ &[g_{[0,2]}, f_{[4,7]}], [g_{[0,2]}, f_{[3,6]}], \\ &[g_{[0,2]}, f_{[3,6]}]\} \end{aligned}$$

So the minimal age of f in the concrete states is given by $d_{(f2)|g}$, because it is lower than the age of f in state $d_{(f1)|g}$. The calculation of the age of f for the abstract may state $\hat{d}_{|f}$ is as follows:

$$\begin{aligned} age_{\hat{d}_{|g}}(f) &= \sigma^{may}_{\hat{d}}(f, g) = \min(age_{d_{(f1)|g}}(f), age_{d_{(f2)|g}}(f)) \\ &= \min(age_{\hat{d}}(g) + \text{size}(g), age_{\hat{d}}(f) + \text{size}(g)) = \min(4, 3) \\ &= 3 \end{aligned}$$

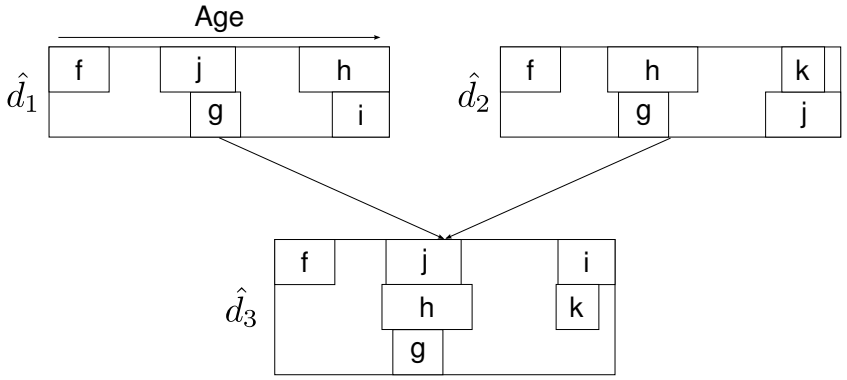


Figure 4.15: Join function for the LRU D–ISP may analysis with f as active function

Then the resulting abstract may states for the LRU D–ISP build by the `update` function defined in Equation (4.55) are:

$$\hat{d}_{|f} = [f_{[0,3]}, g_{[4,6]}]$$
$$\hat{d}_{|g} = [g_{[0,2]}, f_{[3,6]}]$$

They can be also build from set of concrete states $d_{|f}$ and $d_{|g}$ using the abstraction function α^{LRU}_{may} defined in (4.53). ♦

After discussing the `update` function for the may set the `join` function for merging control flows during analysis will be described. It is similar to the `join` for may analysis of the LRU cache discussed in Section 4.2.2.

Definition 13. The `join` function combines two abstract D–ISP states to one:

$$join^{LRU}_{may} : \hat{D}_{may} \times \hat{D}_{may} \mapsto \hat{D}_{may}$$

It creates the joined abstract D–ISP state $\hat{d}_3$ and is defined as:

$$join^{LRU}_{may}(\hat{d}_1, \hat{d}_2) = \hat{d}_3, \quad where\ \forall f \in \hat{d}_1 \vee \hat{d}_2 : age_{\hat{d}_3}(f) = \min\left(age_{\hat{d}_1}(f), age_{\hat{d}_2}(f)\right) \tag{4.59}$$

The join transfers all functions that are in at least one abstract D–ISP state into the new D–ISP state with the minimum age of both entries. Thus the join can be seen as an union of states applying the minimum age for the functions. ■

Example. In Figure 4.15 a join of two arbitrary abstract states is shown for the may analysis. As for the join in must analysis for both memory states the same function is active, which is denoted by the age of 0. In this case it is function f, which keeps its position in the merged state. For all other functions their minimal age is used in the created state $\hat{d}_3$. ♦

Support for the D–ISP Function Tables

To maintain the functions in the scratchpad the D–ISP uses additional helper memories including a *lookup table* and a *mapping table* as introduced in Section 3.2.3. Since these tables have a limited size[6] and on table overrun functions need to be evicted, a proper D–ISP content analysis has to

[6]Both have the same size determined by the parameter no_{func}.

take the size of the management tables into account. Therefore, an additional content analysis of the tables has be done. Then it can be determined, which function is evicted on management table overrun. As for the D–ISP content analysis the analysis of the table content depends on the replacement policy that is used for the management tables. Because all entries of the tables are of the same size, for the analysis of the table content an analysis of a fully associative cache can be employed. The analysis has to consider the size of the maintenance tables as number of cache lines, the function entries as the maintained memory references, and the `update` function has to be only triggered on call and return. To reduce the occurrence of the table overrun it is recommended to employ the same replacement policy for the management table as for the D–ISP itself. So for the analysis of the D–ISP management tables using the LRU replacement policy the analysis for the LRU cache as shown the Section 4.2.2 can be used.

Furthermore, both analysis have to be synchronised, because if a function is evicted from the scratchpad memory or a function is evicted due to management table overrun the abstract must and may states need to be updated for both analyses, i.e. the abstract D–ISP states for the D–ISP memory and the abstract table states for the D–ISPs management tables.

4.3.2 FIFO Replacement Policy

As for the cache the FIFO replacement policy for the D–ISP (denoted as D–ISP$_{FIFO}$) is not easy to be analysed by abstract interpretation. This is because the must analysis depends on the may analysis. For example if a function that is activated is not in the must state, it cannot be inserted in the last-in position, because it might be already in the D–ISP. So the insertion position in the must set depends on the may position of the function. If no may analysis is possible for FIFO, then the must analysis has to insert every activated function to the first-in position, in which it is evicted at the next D–ISP miss. This leads to a correct, but very pessimistic must analysis. Thus, it is not sufficient to analyse the FIFO policy without as proper may analysis.

As shown for the I–Cache$_{FIFO}$ no proper may analysis is found, due to the fact that a set of concrete states cannot be described in an abstract domain using the best and worst position of each memory reference only. This also holds for the D–ISP, since it applies the same replacement policy with the difference that the functions have non-uniform sizes. Then from analysis point of view, the FIFO cache can be seen as the special case of a FIFO D–ISP containing only equally sized functions. Therefore, the generalisation of the FIFO cache analysis for the D–ISP affects the whole analysis and cannot be created by simple adaptations. Such that it is doubtful if the method proposed by Reineke and Grund [2008] to build a safe and more precise may analysis for a FIFO cache by using a larger LRU cache, could be utilised for the D–ISP. For the same reason also other analysis techniques proposed by Grund and Reineke [2009; 2010a] cannot be applied. So an adaptation of FIFO cache analysis methods for a sound D–ISP$_{FIFO}$ analysis or the composition of a custom D–ISP$_{FIFO}$ analysis is too challenging and beyond the scope of this work.

Therefore, a content analysis using abstract interpretation is currently not possible. This is in contrast to the proposed analysis of the *method cache* in [Kirner and Schoeberl, 2007]. The proposed FIFO analysis ignores the fact that the must analysis depends on the may state and using the last-in position on insert, which causes a too optimistic and unsafe must analysis.

Since no other content analysis techniques are known for the FIFO replacement policy (except for model checking that requires a huge amount of computational power), the content analysis of the D–ISP$_{FIFO}$ is done by assigning all concrete memory states to the abstract domain, which is also known as *collecting semantics*. This results in general in a higher analysis overhead since every reachable concrete state has to be maintained and not only two states (for must and may analysis) have to be assumed for each abstract memory state and the analysis complexity may

Algorithm 4.3 Activation of a function in a set of concrete D–ISP$_{FIFO}$ states

Input: Set of D–ISP$_{FIFO}$ states: M ∧ activated function: f
Output: Updated set of D–ISP$_{FIFO}$ states: M

```
for ∀m ∈ M do
  if f ∉ m then
    m ← (m, f) // Add f at the end of the FIFO
    while size(m) > size_max do
      (x, m) ← m // Delete function from the front of the FIFO until function f fits
    end while
    if |m| > no_func then
      (x, m) ← m // Delete oldest function if the adding of f caused a mapping table overflow
    end if
  end if
end for
return M
```

Algorithm 4.4 Joining two sets of concrete D–ISP states

Input: Set of D–ISP$_{FIFO}$ states: M, N
Output: Joined set of D–ISP$_{FIFO}$ states: L

```
L ← M
for ∀n ∈ N do
  if n ∉ L then
    L ← (L, n) // Add only the concrete states from N that are not already in L
  end if
end for
return L
```

be amplified. On the other hand the advantage of using all possible states is that the analysis is more precise.

For simplicity of the analysis it is initialised with an empty memory state on start. Otherwise all possible prior states have to be assumed. The algorithm that is used to update all possible concrete D–ISP states that can be assigned to one point in the applications control flow is shown in Algorithm 4.3. It has to be executed for the set of all possible concrete states on call and return of every function. The algorithm also takes the size of the *lookup* respectively *function table* (no_{func}) into account and evicts the first-in function on table overrun. Control flow joins are modelled during content analysis by merging all concrete states from the branches into one set of concrete states. This is shown in Algorithm 4.4. To quantify the overhead of this analysis approach in Chapter 6 its effort is compared to the D–ISP$_{LRU}$ analysis.

The hit and miss categorisation is done by checking all concrete D–ISP states for an abstract state. If the activated function is in all concrete states, the function activation is a hit, thus the activation is categorized as *always hit* (AH). If the function activation results in misses for all possible concrete states, the function activation is classified as *always miss* (AM). Otherwise the activated function is in some of the concrete states but not in all, thus it is *not classified* (NC).

The content of the D–ISP has to be cleared on analysis start. Just assuming an empty memory is not safe for the FIFO replacement policy, because it is not the worst possible initial state [Grund and Reineke, 2010a]. Therefore, the D–ISP provides a flush instruction, which invalidates the content on application start (refer to Section 3.3.4). The need for an empty

scratchpad holds also for the stack-based policy, since an unknown memory state might also be worse than an empty one.

4.3.3 Stack-Based Replacement Policy

The stack-based replacement policy for the D–ISP (denoted as D–ISP$_{STACK}$) that was introduced in Section 3.2.4 is to be analysed differently than LRU or FIFO. This is because the decision what to evict depends on the activation type of the function which is either call or return. By this the replacement policy tries to hold the active branch of the call tree in the memory, as long as possible. So it favours to keep the caller functions rather than other leaves of the call tree in memory, which might be accessed recently. This can be beneficial, because the application will always return to the caller function. Due to the fact that the stack-based replacement policy can be assumed as a FIFO policy with two counterrotating eviction pointers, it suffers the same problems regarding its analysability as the FIFO replacement policy.

So as for the D–ISP$_{FIFO}$ no proper the content analysis using abstract interpretation was found. Therefore, it is necessary to track all possible concrete D–ISP memory states during content analysis. This has the same impact on analysis complexity and memory usage as the analysis of the FIFO replacement policy shown in the previous section. The difference is that the replacement of functions in the concrete states is done differently. To maintain the content of all possible concrete states during analysis an `update` function for call and return is given in the Algorithms 4.5 and 4.6, respectively. Both algorithms use the set of all possible concrete memory state, the activated function f, and the function that was activated before p[7] to create the updated set of the concrete D–ISP states. The shown algorithms also respect the size of the *lookup* and *mapping table* (no_{func}), such that on table overrun the function with the highest stack distance (which is marked by the *Call* respectively *Return Eviction Pointer*) is evicted. On control flow join the set of the maintained concrete states is merged for all joined branches. For the stack-based replacement policy the same algorithm for the join as for the FIFO replacement policy is used, which is shown in Algorithm 4.4.

Example. The Figure 4.16 shows how the concrete D–ISP states are split and merged during analysis. On a control flow branch the set of D–ISP states that is valid on the branch is propagated to every control flow. If multiple control flows are merged, e.g. after returning from a branched control flow, the concatenation of the D–ISP states from the branches build the merged D–ISP memory state. If a D–ISP memory state consists of multiple concrete states, each state has to be updated on every function activation. ⧫

The categorisation of the function activations in *always hit/miss* or *not classified* during the application on call or return is similar as for the D–ISP$_{FIFO}$.

4.3.4 Sensitivity of the D–ISP Memory on Unknown States

In this section the sensitivity of the different D–ISP replacement policies on unknown states is briefly discussed. The sensitivity of a memory analysis on an unknown memory state is of importance, because on application start the content of a memory is usually not known. If an unknown initial memory state complicates a safe and precise WCET analysis, the unknown memory content has to be transferred into a known one, e.g. by initiating a memory flush.

Furthermore, the sensitivity on unknown memory states is also of interest for an analysis of an application with different tasks that share the D–ISP. The tasks may be executed in unknown order, consign an ambiguous memory state to their successors, or can be interrupted by other

[7] Depending on the activation type this is either the caller or the callee function.

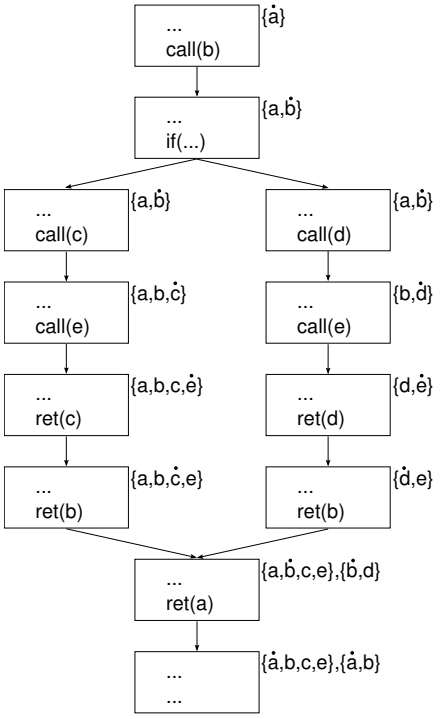


Figure 4.16: Example for using all reachable concrete states for D–ISP$_{STACK}$ analysis

tasks. Hence, the analysis may not know the content of the D–ISP on task (re-)activation. Therefore, the impact of an unknown memory content on the analysis of the different replacement policies needs to be examined. So if the host processor supports task switches, it is of importance that the D–ISP content analysis is not affected by a D–ISP content build by any other task before. Thus if the replacement policy used by the D–ISP is insensitive to any unknown prior state, the memory content on task switch can be ignored. Otherwise, if the behaviour of the replacement policy may depend on D–ISP content provided by the task that was interrupted or previously executed, the content analysis cannot be precise or even correct and thus the unknown memory state has to be cleared on task switch.

In the following examples are provided showing the sensitivity of the replacement policies of the D–ISP on unknown memory states. A formal proof is not given within this work. For cache replacement policies the sensitivity on unknown memory states is investigated and proven by Reineke et al. [2007]. By an adaptation of this work the sensitivity of the D–ISP's replacement policies on an unknown memory state is examined.

LRU Replacement Policy

The LRU replacement policy is not negatively affected by prior memory states. So from analysis point of view on a task switch an empty D–ISP memory can be assumed. This is safe, because the LRU replacement policy always sets the minimal age to functions on their access. Thus, any prior memory state does not negatively affect the execution of functions, because functions that are not accessed will be evicted from the scratchpad without interfering the eviction order of other functions.

Algorithm 4.5 Activation of a function in a set of concrete D–ISP$_{STACK}$ states for calls

Input: Set of D–ISP$_{STACK}$ states: M $\wedge$ activated function: f $\wedge$ caller function: p $\wedge$ activation type: $a = CALL$

Output: Updated set of D–ISP$_{STACK}$ states: M

```
for ∀m ∈ M do
  size_free ← size_D–ISP − size_used_m // Get the free size for state m
  if f ∉ m then
    count ← 0 // Number of bytes written for f
    (m1, p, m2) ← m // Split the memory into functions before and after p
    while count < size(f) ∧ m2 ≠ ∅ do
      (x, m2) ← m2 // Evict functions on the right to the previous activated one
      count ← count + size(x)
    end while
    count ← count + size_free // If all right hand functions are evicted, use the free space
    while count < size(f) ∧ m1 ≠ ∅ do
      (x, m1) ← m1 // Evict the caller functions with the highest stack distance
      count ← count + size(x)
    end while
    if count ≥ size(f) then
      m ← (m1, p, f, m2) // Add the function f right after the previous activated one
    else
      m ← (f) // If f is too large to allow keeping p, m contains f only
    end if
    if |m| > no_func then
      // Eviction on mapping table overflow:
      (m1, f, m2) ← m
      if m1 ≠ ∅ then
        (x, m1) ← m1 // Evict the function x with the highest stack-distance on call
      else
        (m2, x) ← m2 // Evict the function x with the highest stack-distance on call
      end if
      m ← (m1, f, m2)
    end if
  end if
end for
return M
```

Algorithm 4.6 Activation of a function in a set of concrete D–ISP$_{STACK}$ states for returns

Input: Set of D–ISP$_{STACK}$ states: M $\wedge$ activated function: f $\wedge$ callee function: p $\wedge$ activation type: $a = RETURN$

Output: Updated set of D–ISP$_{STACK}$ states: M

```
for ∀m ∈ M do
  size_free ← size_D–ISP − size_used_m // Get the free size for state m
  if f ∉ m then
    count ← 0 // Number of bytes written for f
    (m1, p, m2) ← m // Split the memory into functions before and after p
    while count < size(f) ∧ m1 ≠ ∅ do
      (m1, x) ← m1 // Evict functions on the left hand side of the previous activated one
      count ← count + size(x)
    end while
    count ← count + size_free // If all left hand functions are evicted, use the free space
    while count < size(f) ∧ m2 ≠ ∅ do
      (m2, x) ← m1 // Evict callee functions with the highest stack distance
      count ← count + size(x)
    end while
    if count ≥ size(f) then
      m ← (m1, f, p, m2) // Add the function f right after the previous activated one
    else
      m ← (f) // If f is too large to allow keeping p, m contains f only
    end if
    if |m| > no_func then
      // Eviction on mapping table overflow:
      (m1, f, m2) ← m
      if m2 ≠ ∅ then
        (m2, x) ← m2 // Evict the function x with the highest stack-distance on return
      else
        (x, m1) ← m1 // Evict the function x with the highest stack-distance on return
      end if
      m ← (m1, f, m2)
    end if
  end if
end for
return M
```

Example. Assume the arbitrary concrete D–ISP memory state containing a set $X = \{x^0, \cdots x^n\}$ of n functions:

$$[x^0, \cdots x^n]^{LRU}_{\text{size}(D\text{–}ISP)} \text{ with } \sum_{i=0}^{n} \text{size}(x^i) \leq \text{size}(D\text{–}ISP)$$

On accessing a set of m functions $Y = \{y^0, \cdots y^m\}$ in the order $\langle y^0, y^1, \cdots y^m \rangle$, which have a size larger than the scratchpad:

$$\sum_{i=0}^{m} \text{size}(y^i) \geq \text{size}(D\text{–}ISP)$$

and the two sets X and Y are not disjoint:

$$X \cap Y \neq \emptyset$$

Then the D–ISP content will contain only a subset of the functions from Y:

$$[x^0, \cdots x^n]^{LRU}_{\text{size}(D\text{–}ISP)} \xrightarrow{\langle y^0, y^1, \cdots y^m \rangle} [y^m, y^{m-1}, \cdots y^j]^{LRU}_{\text{size}(D\text{–}ISP)}$$
$$\text{with } j \text{ holds } \sum_{i=j}^{m} \text{size}(y^i) \leq \text{size}(D\text{–}ISP) \text{ and } \sum_{i=j-1}^{m} \text{size}(y^i) > \text{size}(D\text{–}ISP)$$

This holds because the position of a function in the memory set is the same, no matter if this function was in the D–ISP or not. Consider these two cases:

$$[x^0, \cdots z, \cdots x^a]^{LRU} \xrightarrow{z} [z, x^0, \cdots x^a]^{LRU}$$
$$[x^0, \cdots x^b]^{LRU} \xrightarrow{z} [z, x^0, \cdots]^{LRU}$$

So for the LRU replacement policy the resulting memory state is equal. Notice that the type of activation of the functions in $\{y^0, \cdots y^m\}$ is not of importance. ♦

With no information on the previous content a LRU policy is able to gain the content information very fast, such that in the worst case it can be assumed that the D–ISP is empty, if no information is available. This is similar to an LRU cache as Reineke et al. [2007] showed. Hence, the D–ISP$_{LRU}$ is insensitive to unknown states. Thus on task switch the content analysis can assume an empty D–ISP memory, whereas the D–ISP may contain an unknown set of functions. So flushing the D–ISP$_{LRU}$ on task switch is not necessary.

FIFO Replacement Policy

The FIFO replacement policy is sensitive to unknown states, because it does not alter the position of memory entries on their access. Therefore, an information regain of the D–ISP content is not assured by activating a function that might be in the D–ISP. The following example shows that the position of the functions in D–ISP$_{FIFO}$ depends on older states of the D–ISP content.

Example. Assume the arbitrary concrete D–ISP memory state containing a set $X = \{x^0, \cdots x^n\}$ of n functions:

$$[x^0, \cdots x^n]^{FIFO}_{\text{size}(D\text{–}ISP)} \text{ with } \sum_{i=0}^{n} \text{size}(x^i) \leq \text{size}(D\text{–}ISP)$$

Accessing a set of m functions $Y = \{y^0, \cdots y^m\}$ in the order $\langle y^0, y^1, \cdots y^m \rangle$ for which the following equation holds:

$$\sum_{i=0}^{m} \text{size}(y^i) \geq \text{size}(D\text{–}ISP)$$

Then the D–ISP content will contain only a subset of the functions from Y:

$$[x^0, \cdots x^n]^{FIFO}_{\text{size}(D\text{–}ISP)} \xrightarrow{\langle y^0, y^1, \cdots y^m \rangle} [y^m, y^{m-1}, \cdots y^j]^{FIFO}_{\text{size}(D\text{–}ISP)}$$

$$\text{with } j \text{ holds } \sum_{i=j}^{m} \text{size}(y^i) \leq \text{size}(D\text{–}ISP) \text{ and } \sum_{i=j-1}^{m} \text{size}(y^i) > \text{size}(D\text{–}ISP)$$

The eviction of the functions from X is guaranteed by the fact that any inserted function of Y moves all present functions towards the first-in position from which they are evicted, if the D–ISP size exceeds. This holds only if the two function sets are disjoint:

$$X \cap Y = \emptyset$$

This restriction is of importance, since the position of a function after its access, depends on the D–ISP content before:

$$[x^0, \cdots z, \cdots x^a]^{FIFO} \xrightarrow{z} [x^0, \cdots z, \cdots x^a]^{FIFO}$$
$$[x^0, \cdots x^b]^{FIFO} \xrightarrow{z} [z, x^0, \cdots]^{FIFO}$$

So in both cases the function z will in a different position showing that the position of a function after its activation in the D–ISP$_{FIFO}$ depends on the D–ISPs content before its activation. Furthermore, also the position or presence of the other functions is affected: e.g. x^0 is either in the last-in position or it was displaced to the next position in the FIFO queue. ⧫

Therefore, the behaviour of FIFO replacement policy depends on previous memory states, such that a worst-case behaviour cannot be assumed, if no information are available about the previous state. So on application start and any task switch the flushing the memory content by the D–ISP controller is mandatory to allow a safe content analysis. Notice that if it can be ensured that tasks never share a function, the flushing of the D–ISP content is only necessary on application start. Reineke et al. [2007] inspected the sensitivity on an unknown memory state for a FIFO cache and are also showing that its behaviour can be affected by prior memory states.

Stack-Based Replacement Policy

Because the stack-based replacement policy can be represented by a combination of two counter-rotating FIFO queues, it is also sensitive to prior memory states. The difference of the stack-based replacement policy and FIFO is that the insertion position of a function on a miss depends on its activation type. As the following example shows the dependence of the D–ISPs content on previous memory states holds for both function activation types, call and return.

Example. Assume the two following concrete D–ISP memory states in which the function x^j calls the function y.

$$[x^0, \cdots, \dot{x}^j, \cdots y, \cdots x^n]^{STACK} \xrightarrow{x^j \text{ calls } y} [x^0, \cdots, x^j, \cdots \dot{y}, \cdots x^n]^{STACK}$$
$$[x^0, \cdots, \dot{x}^j, \cdots x^n]^{STACK} \xrightarrow{x^j \text{ calls } y} [x^0, \cdots x^j, \dot{y}, \cdots]^{STACK}$$

Depending on the presence of y in the D–ISP content before calling y the resulting D–ISP content is different. The same also holds for returning of a function x^j to its caller z:

$$[x^0, \cdots, \dot{x}^j, \cdots z, \cdots x^n]^{STACK} \xrightarrow{x^j \text{ returns to } z} [x^0, \cdots, x^j, \cdots \dot{z}, \cdots x^n]^{STACK}$$
$$[x^0, \cdots, \dot{x}^j, \cdots x^n]^{STACK} \xrightarrow{x^j \text{ returns to } z} [x^0, \cdots \dot{z}, x^j, \cdots]^{STACK}$$

⧫

It is shown that on function activation the eviction order will be different, if the function which is accessed is already in the D–ISP or not. Therefore, as also for the FIFO replacement policy, a flush of the D–ISP content is required on task switch and when the memory state of the D–ISP is unknown. Only if the D–ISP state is known, the D–ISP content analysis for the stack-based replacement policy can be safe and precise. Hence, on application start the D–ISP has to be flushed to ensure a safe WCET analysis.

Chapter 5

Toolchain for the Dynamic Instruction Scratchpad

This chapter provides an insight to the toolchain needed by the D–ISP. It consists of an instrumentation and an analysis tool. The instrumentation tool is needed to add information of the function sizes into the code, such that the D–ISP is able to load them correctly into the scratchpad. The determination of the WCET estimate for a system employing the D–ISP requires its content analysis. Therefore, a WCET analysis framework including D–ISP analysis is also proposed in this chapter.

The Section 5.1 introduces instrumentation tools for the two methods described in Section 3.3 to provide the sizes of the functions to D–ISP controller on their load. In Section 5.2 a timing analysis tool for the D–ISP and other competing instruction memories will be briefly described. It builds the foundation for the WCET evaluation of the D–ISP in the next chapter. The employed methods for timing analysis and for the low-level analysis of instruction memories were introduced Section 2.6 and Chapter 4, respectively. The supported processor architecture of the presented tools is the CarCore processor. In addition to the toolchain the Section 5.3 provides guidelines to allow the effective use of the D–ISP. It also examines code styles that are prohibit by the D–ISP.

5.1 Function Size Instrumentation

The D–ISP has to detect the function size to load it correctly into the memory. Without providing the size of the function the D–ISP controller cannot determine the function's end and thus it is not able to load it. So uninstrumented code cannot be executed with the D–ISP. Hence, the function size instrumentation is mandatory, if an application intends to use the D–ISP. Two methods to allow the D–ISP controller to handle functions correctly were discussed in Section 3.3.4:

- **FDBP**: A Function Delimiter Bit Pattern is placed after every function, allowing to detect its end by the D–ISP **while** loading it.
- **FLE**: A Function Length Encoding instruction is inserted as first instruction of every function, allowing the determination of the function length **before** loading it.

In the following both approaches are described from instrumentation tool point of view. Based on this description a short overview of the implementation of two instrumentation tools is given.

Function Delimiter Bit Pattern Instrumentation

The placement of the FDBP after the end of each function allows the D–ISP controller to detect the end of function when loading it. The FDBP has to fill at least one fetch block to allow an unambiguous and easy detection of the pattern by the D–ISP controller. Because the TriCore ISA that is used by the CarCore processor features instructions with different lengths, different alignments of the last instruction of a function are possible. Therefore, the length of the bit pattern has to be larger than one fetch block, but can be bounded by:

$$\text{size}(FDBP) \leq 2 \cdot \text{size}(fetch\_block) - \min(\text{size}(instr))$$

In the CarCore processor architecture the fetch blocks have a size 64 bit and the smallest instruction in the TriCore ISA [TriCore, 2007b] is 16 bit long. So the size of the FDBP is given as:

$$\begin{aligned} \text{size}(FDBP) &\leq 2 \cdot 8\,\text{B} - 2\,\text{B} \\ &\leq 14\,\text{B} \end{aligned}$$

The FDBP itself is chosen to consist of byes with the value `0x55`. Because this value corresponds to no valid opcode in the TriCore ISA, it cannot be accidentally mistaken by the D–ISP controller. Moreover, the FDBP is longer than the maximum size of an instruction, which is 32 bit. Thus it cannot be mixed up with any arbitrary instruction containing a part of the pattern as offset or displacement value. Hence, the FDBP is defined by 14 bytes with the value `0x55`.

Using the size for FDBP shown in the equation above, it is ensured that it will always fill at least one fetch block and thus the function end can be easily determined by the D–ISP, as the following example clarifies:

Example. Assume that "`----`" marks any 16 bit instruction or a part of any other instruction and "`5555`" represents one of the 7 `0x5555` 16 bit words that build the FDBP. Furthermore, "`RRRR`" denotes the 16 bit `return` instruction of a function, which is the last instruction of the function. Since the minimum instruction size is 16 bit, the following alignments of the `return` instruction and therefore also of the FDBP are possible:

```
|   Fetchblock x    | |   Fetchblock x+1  | |   Fetchblock x+2  |
|RRRR|5555|5555|5555| |5555|5555|5555|5555| |----|----|----|----|
|----|RRRR|5555|5555| |5555|5555|5555|5555| |5555|----|----|----|
|----|----|RRRR|5555| |5555|5555|5555|5555| |5555|5555|----|----|
|----|----|----|RRRR| |5555|5555|5555|5555| |5555|5555|5555|----|
```

Using the length of 14 B for the FDBP, always at least one fetch block consists of the unambiguous part of the FDBP only. This allows the detection of the function end by comparison of a complete fetch block without taking the alignment of the FDBP into account. ⧫

Notice that it is impossible for the host processor to execute the FBDP, because on execution of the `return` the control flow is changed to the caller of the function. Therefore, the presence of FDBP cannot affect the state of the processor or the application.

Function Length Encoding Instrumentation

Using the function length encoding a FLE instruction is placed on the very beginning of every function to instrument. This instruction holds the length of the function and thus allows the

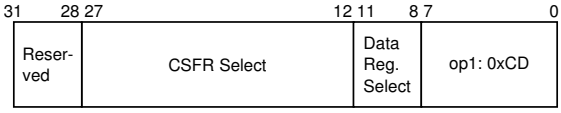


Figure 5.1: Decoding of the TriCore MTCR instruction, see [TriCore, 2007b, p. 2-323] for details

D–ISP controller to load the necessary number of instructions. Therefore, a dedicated instruction is needed in the host processor. In the CarCore processor that implements the TriCore ISA the MTCR (Move To Core Register) instruction is used for encoding the FLE instruction. The MTCR instruction usually moves a value from data register into a Core Special Function Register (CSFR) of the TriCore [TriCore, 2007b]. The structure of the instruction is shown in Figure 5.1. It has a width of 32 bit, the CSFR is selected by a 16 bit, value and the data register by 4 bit. To classify the MTCR instruction as FLE instruction unused CSFRs were used to identify the FLE instruction. Thus the FLE detection bit mask is `0x0e0000cd`. To encode the function length 2 B of the instruction (bit 23 to 7) are used. The function length is encoded in bytes resulting in a maximum function size of $2^{16} = 65,535$ B. Functions larger than 64 KiB cannot be instrumented using this FLE instruction. But in embedded systems an upper bound for a function size of 64 KiB seems to be sufficient for nearly all applications. Anyhow, larger functions can be split to allow their instrumentation. Furthermore, it is also possible to encode the size in a more coarse granularity (e.g. as 32 bit words instead of bytes). This would allow larger function sizes in the FLE instruction.

To ease the detection of the FLE instruction by the D–ISP controller in hardware the instruction should be aligned to the beginning of a fetch block. In the CarCore implementation the fetch width is 64 bit, resulting in aligning the FLE instruction to 64 bit addresses. The function (and thus FLE instruction) alignment can be done by the linker on application build or later by the instrumentation tool.

The instrumentation of the application that is to be executed in a system with the D–ISP can be done in two ways: first by instrumenting the code during compilation and second by instrumenting the executable code. Both variants are briefly described in the following.

5.1.1 Compile-Chain Instrumentation

Instrumenting the function size information for the D–ISP during application build needs to be done during compilation e.g. on the intermediate assembler code. This is required, because the restrictions of the placement of the added instruction or bit pattern cannot be enforced in a higher level programming language (like C): It cannot be ensured that the first instruction of a function is the FLE instruction, because the compiler may add instructions before the first high level programming language statement of a function, e.g. to initialise local variables. For the instrumentation with FDBP, the bit pattern cannot be placed at the end of the function, because after the return instruction the compiler usually omits the unreachable code. Therefore, the instrumentation has to be done e.g. on the intermediate assembler code created by the compiler. The Figure 5.2 depicts a build chain including the instrumentation of the code when building an application. Here a special instrumentation tool is used, but it is also possible to instrument the code by the compiler as e.g. Falk and Lokuciejewski [2010] proposed.

The instrumentation tool has to parse the generated assembler code and identify the functions within the code. For instrumentation with FDBP the function end of each function (or the

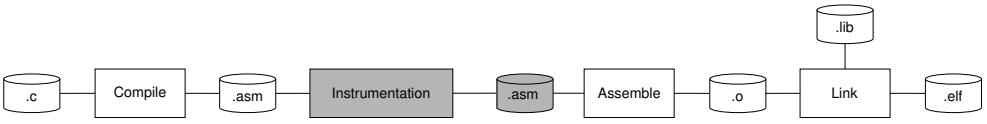


Figure 5.2: Compile-chain instrumentation

beginning of the following function) is detected and then the bit pattern is inserted as data by using assembler directives.

When using the FLE instruction, the instrumentation tool has to parse the assembler code and insert the FLE instruction at the very beginning of each function. Therefore, it needs the function's size, which easily can be determined by output of the build toolchain. But the instrumentation tool has to be aware of that the additional instruction enlarges the function. To enforce that each function is aligned to 64 bit addresses the instrumentation tool adds an alignment assembler directive before the FLE instruction. After the updated assembler code is generated it is assembled and linked to the executable application.

A drawback of the compile-chain instrumentation is that precompiled library code cannot be instrumented. Thus the D–ISP can be used only for application code that is available to the application maintainer. Any library functions for which the source code is not present, as for the operating system or code from commercial suppliers, cannot be instrumented. To address this problem the post-link instrumentation may be used.

5.1.2 Post-Link Instrumentation

The post-link instrumentation allows to add the function size information needed to use the D–ISP into the executable without having the sources or needing to compile it. Therefore, the instrumentation tool determines the start and end point of each function the executable, which is e.g. available as ELF[1] or RAW file. Also the sizes for every function is extracted from the executable file. So the instrumentation tool can add the FDBP or the FLE instruction at every function end or begin, respectively.

Due to the code in the executable is already linked the instrumentation of the function size information can require the relocation of other code or data parts within the executable. Therefore, the post-link instrumentation needs to check and eventually adjust the addresses of code and data sections after adding the FLE instruction or the FDBP. Furthermore, the target addresses or offsets of `jump` and `call` instructions may need to be adjusted. It is also possible to align the FLE instructions to 64 bit addresses during the instrumentation, if the functions are not already aligned by the linker. After instrumentation the updated application can be executed using the D–ISP. How the instrumentation tool is embedded in the application build toolchain is depicted in Figure 5.3.

For the post-link instrumentation the instrumentation tool is aware of the concrete alignment of each instruction. Thus, it does not necessarily need to add the complete FDBP. It is sufficient that one complete fetch block with the FDBP is written, to allow the D–ISP to detect the end of the function. This reduces the overhead added by instrumentation. The amount of reduction depends on the alignment of the last instruction in the function. Anyhow, the minimal size of the FBDP cannot be less than the size of one fetch block, because the D–ISP controller compares for complexity reasons always the complete fetch block. Hence, the minimal overhead is 8 B per function.

[1] Executable and Linking Format

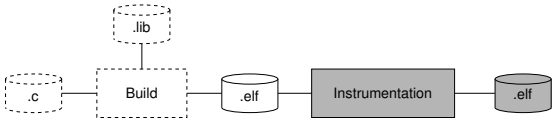


Figure 5.3: Post-link instrumentation

The great advantage of the post-link instrumentation is that the code of the application does not need to be available, which is of importance when libraries that are not available as source code are used. On the other hand the complexity of the post-link instrumentation is magnitudes higher than for the compile-chain instrumentation, but due to its support of libraries it is worth the effort. The post-link instrumentation tool is integrated into the timing analysis tool that is discussed in the Section 5.2, because both tools process the linked application code and have to rebuild its structure.

5.1.3 Quantification of the Instrumentation Overhead

The FDBP approach adds the FDBP for each instrumented function. Thus the application code is increased for each function by:

$$\begin{aligned} \text{overhead}(FDBP) &\leq 2 \cdot \text{size}(fetch\_block) - \min(\text{size}(instr)) \\ &\leq 14\,\text{B} \\ \text{overhead}(FDBP) &\geq \text{size}(fetch\_block) \\ &\geq 8\,\text{B} \end{aligned}$$

The instrumentation using the FLE instruction increases the code size for each function by the size of the FLE instruction which is 32 bit. Furthermore, to allow a detection of this instruction by the D–ISP controller every function has to be aligned to 64 bit addresses. The overhead caused by the alignment depends on the position of the function in the memory, but it is at most 6 B. So the maximum and minimum overhead for the FLE instrumentation per function can be determined by:

$$\begin{aligned} \text{overhead}(FLE) &\leq (\text{size}(fetch\_block) - \min(\text{size}(instr))) + \text{size}(FLE) \\ &\leq (8\,\text{B} - 2\,\text{B}) + 4\,\text{B} \\ &\leq 10\,\text{B} \\ \text{overhead}(FLE) &\geq \text{size}(FLE) \\ &\geq 4\,\text{B} \end{aligned}$$

Thus the instrumentation with the FLE instruction causes less overhead than the FDBP approach, even if the functions have to be aligned. Moreover, the FLE instruction allows the reverse function load needed by the stack-based replacement policy. Hence, in the following the FLE instructions are used for the function size determination in the D–ISP controller.

5.2 Static Timing Analysis Tool for the D–ISP

The quantification of the impact of the D–ISP on the WCET requires a timing analysis of the processor and the D–ISP. In Section 2.6 different approaches to obtain a WCET estimate are

presented. Because of the advantages of a static analysis, for which no complete or worst-case input set and worst-case initial system state has to be determined, a timing model of the D–ISP and the host processor was implemented in a static timing analysis tool: The *Instruction Scratchpad Timing Analysis Program* (ISPTAP) supports the program flow analysis for TriCore binaries, the pipeline analysis for the CarCore processor, and memory analysis for the D–ISP and other common instruction memories in embedded systems. The low-level analysis for the supported instruction memories was already discussed by Chapter 4 in detail. The proposed ISPTAP tool is targeted to the analysis of simple processor architectures (without branch prediction, prefetching, out-of-order execution, or any other speculation) and only provides a subset of common methods of static timing analysis that are useful for the instruction memories under observation. So the feature set of ISPTAP is much smaller than for commercial or research WCET tools like aiT [Heckmann and Ferdinand, 2006] or OTAWA [Ballabriga et al., 2011], which support e.g. data memory analysis including address analysis for data accesses, out-of-order execution, branch predictor analysis, loop bound determination, recursion, run-time optimisations (for large applications), and IDE integration. The ISPTAP tool was developed in C++ bearing in mind to easily replace or enhance parts/steps of the analysis and extend the supported memory types. An overview of the ISPTAP tool is depicted in Figure 5.4 showing the data flow of the WCET analysis. The classical division of a static analysis consists of the three steps: *program flow analysis*, *low-level analysis* and *WCET calculation* as e.g. described in [Engblom et al., 2003]. In the figure the low-level analysis is split into pipeline analysis and instruction memory analysis to show that these steps are separated and to highlight the main focus of the ISPTAP tool, which is the instruction memory analysis. These four steps performed by ISPTAP during static timing analysis are shortly described as:

- **Program Flow Analysis**: The executable of the program under analysis is split into basic blocks, connected in a control flow graph, and enriched with loop bounds to model the possible flow through the application.
- **Pipeline Execution Cost Analysis**: For each basic block the execution cost is determined without taking the memory system into account.
- **Instruction Memory Cost Analysis**: Depending on the memory type that is used in system configuration the cost caused by the instruction memory system is determined and added to the basic blocks as memory penalty.
- **WCET Calculation**: By finding the worst-case path of the application taking the cost of pipeline and memory for each basic block and the program structure into account the WCET estimation is calculated.

The analysis steps are described briefly in the following sections to get insight in the employed well-known techniques and their implementation in a concrete evaluation framework for the proposed D–ISP. Also a brief validation of the employed pipeline timing model is presented. The results of the WCET analysis by the ISPTAP tool will be discussed separately in Section 6.2.

5.2.1 Program Parsing and Structural Representation

Executable Parsing

The application that is to be analysed has to be provided as linked executable, because it contains the addresses of data and instructions of the application as it will be run on the system. The knowledge of the addresses is of importance for the timing analysis, because the execution cost of a program is sensitive to the alignment of code and data. During this analysis step the application is

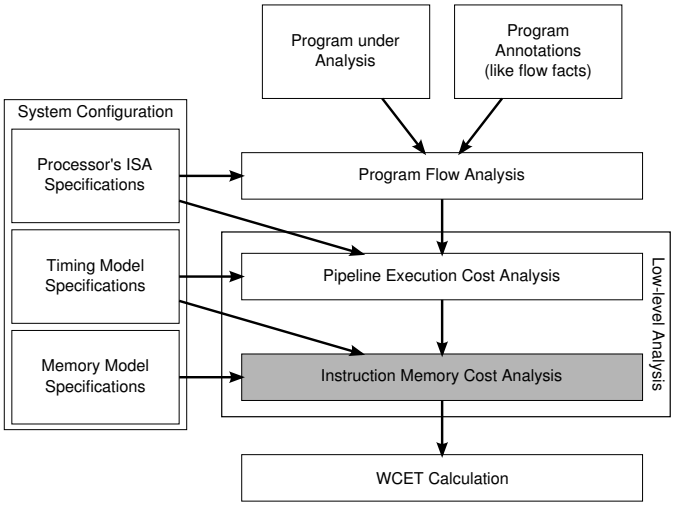


Figure 5.4: Analysis steps and data flow of ISPTAP

split into basic blocks, which will be connected to a control flow graph per function. Calls to other functions are represented as nodes in the control flow graphs. ISPTAP also supports indirect jumps and calls by the help of user annotations that contain their possible target addresses.

Program Representation

To represent the structure of the entire application under observation that contains the entry function (e.g. the main function of the application) and all reachable called functions a so called *supergraph* [Myers, 1981] is created. The supergraph adds called functions as control flow graph only once and enhances them with call and return points depending on the calling context. So a function that is called multiple times from different contexts is represented only once, but it is entered from different call points for the different contexts. This reduces the overall size of the complete control flow graph and still allows the differentiation of multiple calling contexts in the following analysis steps.

Loop Annotation

Loops within the application need to be provided with an upper bound otherwise the WCET cannot be calculated correctly. Thus the application programmer has to provide a file containing all loops and their maximal number of iterations in the application to the ISPTAP tool. This file is similar to the flow facts that are used by OTAWA [Ballabriga et al., 2011]. It would also be possible to automatically extract the maximum iteration count of loops by the analysis of the application as e.g. done in [de Michiel et al., 2008], but this is out of scope for the ISPTAP tool.

5.2.2 Pipeline Execution Cost Analysis

CarCore Instruction Execution Timing Model

As described in Section 3.3.2 the CarCore processor features two pipelines, one for integer instructions and one for address instructions. Two instructions can be issued within one cycle,

Table 5.1: Instruction timing for the CarCore

Instruction	**Processing Time**
Address instruction	1 cycles
Integer instruction	1 cycles
Address instruction (as direct successor of an integer instruction)	0 cycles
Branches	3 cycles
Load (`LD`)	MMAT in cycles
Store (`ST`)	(MMAT - 1) cycles
Call microcode	58 cycles
Return microcode	51 cycles
`LDMST` microcode (Load-Modify-Store)	`LD`+1+`ST` cycles
`SWAP` microcode (Load-Modify-Store)	`LD`+`ST`+1 cycles
`ST.DA` microcode (Store Double-Word)	2 · `ST` cycles
`ST.T` microcode Store Bit	`LD`+1+`ST` cycles
Division is implemented in TriCore ISA by a sequence of `DVINIT`, `DVSTEP` and `DVADJ`, so no microcode is necessary.	

if an address instruction directly follows an integer instruction. Otherwise only one instruction can be issued to one of both pipelines. Complex instructions are implemented in interruptible microcodes consisting of micro-ops [Mische et al., 2010a]. The Table 5.1 shows the cycle count needed to process the different instructions in the pipeline. The value 1 denotes that every cycle one instruction can be issued, 0 represents the case that the instruction is issued in parallel with another instruction. Values higher than 1 denote either microcodes with multiple micro-ops or instructions with additional latency times due to memory access or branch penalty. Notice that in addition to the processing time for the instructions shown in the Table 5.1 misses the latency needed to pass all pipeline stages after issuing. This additional latency has to be considered only on the very last basic block of the application.

For *jumps* a branch penalty is added as extra latency to the execution time. It is needed to determine if a jump is taken or not and to deliver the calculated the target address to the fetch stage. This latency is fixed, because the CarCore does not support branch prediction. Therefore, the penalty has also to be charged for not taken branches to hinder the speculative execution before the branch direction is determined by the execute stage. Thus for any branch instruction 3 cycles are accounted.

If no data memory hierarchy is assumed, *loads* and *stores* are always directed to the off-chip memory. The support of any data memory hierarchy requires an additional address analysis for each memory access. Because the focus of the analysis is on the instruction memory, a support for data memory hierarchy is currently not implemented in the ISPTAP tool.

The processing time of the *load* instruction is given by the Maximum Memory Access Time (MMAT). This is due to the split-phase character of the memory instructions [Mische et al., 2010a]: On issuing the first part of the *load* the pipeline calculates the address and requests the data from memory. The issue stage has to delay the second part of the *load* that writes

the loaded value into a register until the value is delivered, which will happen exactly after the memory latency number of cycles. Then the second part of the *load* arrives at the same cycle in the register write back stage as the memory controller delivers the loaded value. For *stores* the second part of the split-phase instruction is not needed, but the issue stage blocks dispatching further instruction until the *store* is completed. Because the *store* instruction does not need to write register values, the cycle in which the second part of the split-phase instruction would be placed is used by the next instruction. Therefore, the processing time for the *store* instruction is one cycle less than for a *load*.

Based on this timing model the ISPTAP tool is able to determine the execution cost of the instructions contained in a basic block for the CarCore processor. The timing effect between two basic blocks, e.g. if an integer instruction that is at the end of one basic block is the followed by an address instruction, which can be issued within the same cycle, is not addressed by the pipeline analysis. These timing effects have an impact on the overall WCET, but by using pessimistic bounds the proposed timing model does not underestimate those effects. Unfortunately, this will affect the tightness of the WCET estimate. Nevertheless, the timing analysis delivers a safe upper bound of the WCET. For a validation and a discussion of the tightness of the pipeline analysis refer to Section 5.2.6.

CarCore Fetch Timing Model

Beside the basic block execution cost an additional cost that is caused by fetching the instructions of the basic blocks from memory has to be considered. Depending on the instruction memory latency this additional fetch cost affects the overall basic block cost with different impact.

Modelling the instruction fetch is done by calculating the ready time for each instruction. The ready time of an instruction within a basic block depends on the following parameters: the fetch block width, the fetch latency, the distance from the beginning of the basic block, the alignment of the instruction (it is possible that one instruction is stored in two fetch blocks), the instruction window size, and the fill state of the instruction window on entering the basic block.

The fetch block width and the fetch latency determine how many instructions can be fetched per cycle. By also taking the position of the instructions within the basic block into account the ready time of every instruction within a basic block can be calculated. If dynamic memories are used in the system that is analysed, the fetch cost determination assumes that every access will be a hit in the dynamic memory. The cost that is imposed by a miss will be added during the memory cost analysis independently in the next analysis step.

To determine the alignment of the instructions in the instruction memory it is necessary to use the same executable for analysis that will later be used by the system. The correct alignment is of importance, because only small changes of the code alignment, e.g. within a heavily used loop, may cause to a completely different timing behaviour.

The knowledge of the instruction window (IW) size is also important for a precise fetch timing model, because the fetch process is independent of the instruction issuing and execution. The fetch process requests instructions until the IW is filled, then the fetch process is stalled. It is restarted, if the pipeline issues an instruction and an IW entry is cleared. As described in [Mische et al., 2010a] the IW of the CarCore holds 3 64 bit fetch blocks, which contain at least four instructions in sum[2]. Beside the size of the IW, its fill state on entering a basic block is of importance for a precise fetch timing analysis. Therefore, the fetch cost determination distinguishes the case, if a basic block is entered by *jumping* to it or by *continuous addressing*. In the first case the IW will be empty, because CarCore flushes the IW on any taken jump. Allowing jumps to instructions that are already in the IW without flushing the IW, would

[2]Due to alignments to 16 bit addresses four 32 bit instructions can require three 64 bit aligned fetch blocks.

require a more elaborate analysis of the IW fill state. Therefore, the CarCore does not support this feature and always flushes the IW on every jump.

If a basic block is entered by exiting the preceding basic block without jumping (also denoted as *continuous addressing*), the ISPTAP tool is aware of the actual IW fill state left by the preceding block. So the instructions that are contained in the IW on entering the basic block are immediately ready and their issuing is not delayed. For any further instructions in the basic block the ready time is calculated as described above.

Support for Simultaneous Multithreading

The CarCore processor supports simultaneous multithreading and allows the execution of multiple threads with different priorities in parallel. The issuing policy of the CarCore ensures that the highest priority thread will be executed as if no other threads would run. Therefore, the timing model for the ISPTAP tool can ignore any effects of lower priority threads when analysing the highest priority thread. To provide a tight timing analysis of any lower priority thread the analysis would need to know the state of any higher priority tread, which is practically infeasible. On the other hand assuming always the worst-case behaviour of the higher priority threads would result in an infinite estimate, because it is possible that the highest priority thread uses the complete resources: e.g. it requires the complete fetch bandwidth or issues one instruction per cycle into each pipeline. For that reason the timing analysis in ISPTAP is only possible for the highest priority thread.

5.2.3 Instruction Memory Cost Analysis

The memory cost analysis differs for static and dynamic memories. The principals for memory analysis were already discussed in Chapter 4. Therefore, the issue of integrating the presented approaches into the ISPTAP tool to obtain the memory penalty for each basic block is in the focus of this section.

Memory Cost Analysis and Optimization for Static Instruction Memories

Using static instruction memories the assignment of the code to the scratchpad memory is fixed. Hence, it is only necessary to determine which instruction, basic block, or function is in which memory and what is the memory latency of this memory. In the ISPTAP tool the following static memories are distinguished:

- **No first level memory (NO–MEM)**: All instructions are located in an off-chip memory.
- **First level memory only (S–ISP)**: All instructions are located in an on-chip scratchpad and are accessed with minimal latency.
- **Function-based static instruction scratchpad (FS–ISP)**: Selected functions are located in an on-chip scratchpad and can be accessed with minimal latency. The remaining part of the application is located in a slower off-chip memory.
- **Basic-block-based static instruction scratchpad (BBS–ISP)**: Selected basic blocks are located in an on-chip scratchpad and can be accessed with minimal latency. The remaining part of the application is located in a slower off-chip memory.

For the first two memories the memory cost analysis is implicitly done via pipeline execution cost analysis (described in Section 5.2.2) with the appropriate fetch bandwidth used in the fetch timing model.

The memories that allow a static assignment (FS–ISP and BBS–ISP) come in two flavours in the ISPTAP: the knapsack-based assignment and the WCP-sensitive assignment, that delivers an optimal solution. For both approaches the ISPTAP needs to perform the pipeline cost analysis for the case that the code is in the scratchpad (S–ISP) and for the case that the code is in the off-chip memory (NO–MEM). Then the benefit to assign parts of the code to the scratchpad can be calculated by the memory cost analysis. As described in Section 4.1 the knapsack-based assignment requires a WCET analysis including the determination of the WCP before the code can be assigned to the scratchpad. This is not the case for the WCP-sensitive assignment, which is aware of the complete application control flow while finding the best assignment. So the implementation of scratchpad assignment in the ISPTAP tool follows the Algorithms 4.1 and 4.2 presented in Section 4.1.

Using the **FS–ISP** several functions are assigned to an on-chip scratchpad. The assignment of the functions is performed by solving the linear programs proposed in Section 4.1.2. After determining the set of assigned functions, the memory analysis considers for the execution of the selected functions the scratchpad memory access latency. For all other functions the latency for the access to the off-chip memory is taken into account. With this memory timing information from the memory cost analysis the ISPTAP tool calculates the final WCET estimate for a system with FS–ISP. Furthermore, the selected functions need to be relocated to execute them from the scratchpad on the target system. This can be done either by linking the selected functions into the scratchpad during build or enhancing the linked executable application file. The first option requires the insertion of a location *attribute* [HighTec, 2003, p. 29] for each selected function in the source code:

```
#define func_a __attribute__ ((section (".sisp"))) func_a
```

Altering the linked executable requires a post-link code adjustment in which the following steps are required:

1. Assign a FS–ISP memory region and address space to each selected function.
2. Adjust all calls to relocated functions.
3. Create a table describing what code section is to be copied to which address in the FS–ISP.
4. Embed code that copies the selected functions to the FS–ISP by using the table of step 3.
5. Invoke the copy process on system initialisation.

These changes to the code were not integrated to the ISPTAP tool, because the first solution is suitable for evaluating the benefits of the FS–ISP. To overcome timing effects due to changed function alignment, all functions that are copied to the FS–ISP need to have the same alignment as the functions in the memory. Furthermore, also the alignment of the functions located in the off-chip memory has not to be changed by the function relocation. Therefore, all functions are always aligned to 64 bit (fetch block) addresses[3]. Hence, function relocation cannot affect the alignment of any function. Otherwise timing effects caused by alignment changes could prevail over the benefits of the FS–ISP.

The approach for the **BBS–ISP** is similar as for the FS–ISP with two differences: the assignment is not done on the granularity of functions and the relocated basic blocks have to be reconnected to preserve the application's control flow. As shown in Section 4.1.3 basic blocks that enter or leave the scratchpad need to be altered by adding an additional jump or changing

[3]This is done by building the applications with the compiler flag `-falign-functions=8`.

the jump target. Therefore, the different penalties for the additional jumps or changed jump targets as introduced in Equation (4.17) need to be taken into account on calculating the WCET considering a BBS–ISP assignment. Furthermore, due to adding additional jumps the pipeline execution cost of basic blocks that are not altered can be affected: A basic block that was entered by continuous addressing and for which a connecting jump was added at the preceding basic block will now be entered by a jump, causing that the IW will be empty at the beginning of its execution. This results in a larger execution cost for this basic block that has to be considered by ISPTAP. Also the alignment of the basic blocks in the BBS–ISP needs to be respected during assignment to the scratchpad memory. Otherwise an undesired timing behaviour caused by changed alignment could arise. Therefore, it has to be assured that the alignment of the basic blocks copied to the BBS–ISP is kept or the alignment change has to be taken into account by recalculating the pipeline execution cost of the affected basic blocks. For simplicity, the ISPTAP tool assumes that the alignment of a basic block will not change by moving it into the BBS–ISP.

To allow the application making use of the WCET improvements imposed by the BBS–ISP the basic blocks have to be copied into the scratchpad memory before application start. These changes can be done either by compiler as Falk and Lokuciejewski [2010] propose or by post-link code adjustment. For post-link adjustment the steps to alter the application are similar to the relocation of functions for the FS–ISP:

1. Assign a BBS–ISP memory region and address space to each selected basic block or consecutive sequence of basic blocks.
2. Alter the jump targets or add jumps to preserve the control flow (see Figure 4.3 on p. 93).
3. Create a table describing what code section is to be copied to which address in the BBS–ISP.
4. Embed code that copies the selected blocks to the BBS–ISP by using the table of step 3.
5. Invoke the copy process on system initialisation.

At the current state the ISPTAP tool cannot perform these code adjustments to build an WCET-optimised application for the BBS–ISP, but nevertheless it is able to calculate the WCET impact of the usage of the BBS–ISP including all timing penalties needed to preserve the control flow for the relocated basic blocks.

So the ISPTAP tool determines the assignment of the FS–ISP and BBS–ISP, models the timing effects of the scratchpad usage, and delivers a WCET estimate of a system with static instruction scratchpad memories. The foundations for the implementation of the WCET-aware knapsack-based code assignment for static memories were build by Sepp [2009].

Memory Cost Analysis for Dynamic Instruction Memories

The analysis of dynamic memories requires a data flow analysis to keep track of the different contents of the memories during program execution. To improve the results of the analysis the graph on which the analysis is done can be extended by inlining functions and unrolling loops. Therefore, the ISPTAP uses the VIVU approach [Martin et al., 1998] to enhance the applications *supergraph*. This extended graph will be used for the memory analysis to determine the content of the dynamic memory on all paths of the applications. The following dynamic memories are modelled in the ISPTAP tool:

- **Fully associative instruction cache with LRU replacement (I–Cache$_{LRU}$)**: The analysis using the abstract interpretation is described in Section 4.2.2.

- **Fully associative instruction cache with FIFO replacement (I–Cache$_{FIFO}$)**: The analysis using the all possible concrete cache states is described in Section 4.2.3.
- **Direct mapped instruction cache (I–Cache$_{DM}$)**: The analysis using abstract interpretation is described in Section 4.2.4.
- **Dynamic instruction scratchpad with LRU replacement (D–ISP$_{LRU}$)**: The D–ISP with LRU replacement policy is described in Section 3.2.4 and its analysis using abstract interpretation is introduced in Section 4.3.1.
- **Dynamic instruction scratchpad with FIFO replacement (D–ISP$_{FIFO}$)**: The D–ISP with FIFO replacement policy is described in Section 3.2.4 and its analysis using the complete set of all possible concrete D–ISP states is introduced in Section 4.3.2.
- **Dynamic instruction scratchpad with stack-based replacement (D–ISP$_{STACK}$)**: The D–ISP with stack-based replacement policy is described in Section 3.2.4 and its analysis using the complete set of all possible concrete D–ISP states is introduced in Section 4.3.3.

The LRU and FIFO cache analysis supports only fully associative caches. Thus the results provided by ISPTAP are not affected by the separation of the instructions into the different sets. However, an analysis of set associative caches can be implemented by analysing multiple fully associative cache sets independently as e.g. described by Ferdinand and Wilhelm [1998; 1999]. The ISPTAP tool allows the selection of arbitrary values for the cache line size and the number of cache lines, although these numbers are typically only powers of two when implementing the caches in hardware. The analysis of configurations that cannot be efficiently implemented in hardware allows a better comparability of the cache memories and the scratchpad memories, since then the choice of the memory sizes in an evaluation does not depend on any restrictions of a hardware implementation. Beside the different replacement policies the design parameters of the D–ISP are the size of the scratchpad, the number of entries in the management tables, and the block size of the scratchpad memory.

After data flow analysis, the obtained memory penalty caused by cache or D–ISP misses is assigned to the basic blocks. For the D–ISP these memory penalties can occur only on calling a and returning to a function, whereas for caches a memory penalty may be assigned to every basic block.

5.2.4 Off-Chip Memory Cost

For simplicity the ISPTAP tool assumes for any off-chip memory access a constant latency. This assumption holds for SRAMs. In contrast to SRAMs the memory latency of DRAMs depends on the state of the memory controller: The DRAM memory is organised in rows and columns [Jacob et al., 2007]. To access a certain word in the DRAM the addressed row has to be *activated*, which triggers the copying of the selected row into a row buffer. Then the memory word can be accessed via column *reads* or *writes*. To obtain multiple memory words a *burst access* can read/write multiple columns within the active row. To access a memory word in another row the active row has to be *precharged*, i.e. the content of the row buffer is written back into the memory. Beside the timing of the memory access in a DRAM additional latency caused by refreshing the memory has to be assumed. This is because the DRAM consist of capacitors, that lose their charge due to leakage current. So the bit information is lost, if the memory cells are not periodically refreshed. For a more detailed DRAM model including the different delays see [Jacob et al., 2007].

Therefore, the memory access latency in DRAMs is not constant, as Bhat and Mueller [2010] show for a real-world example. The ISPTAP tool can model DRAM accesses by assigning a constant safe upper bound for the DRAM access latency, assuming that the full memory access cycle including a refresh delay is charged for every single memory access. This leads to an overestimation of the memory latency, because row access delays for *activation* and *precharge* are considered for every memory access.

To improve the preciseness of the ISPTAP tool regarding DRAM memory accesses it could apply one of the following optimisations. It would be possible to use a custom predictable memory controller that gives timing and bandwidth guarantees, as e.g. proposed by Akesson et al. [2007]. Using common DRAMs the overhead caused by the periodic refresh can be considered in the WCET analysis for example by the approaches of Atanassov and Puschner [2001] or Bhat and Mueller [2010]. It is also possible to model the behaviour of the DRAM by a low-level analysis, as Bourgade et al. [2008] propose. Refer also to Section 2.5 for a more detailed discussion of DRAMs in real-time systems.

5.2.5 WCET Estimation

Using the structural representation of the application, the timing cost of the execution of each basic block in the pipeline, and the additional memory penalties (either caused by static or dynamic memories) the calculation of a WCET estimate is possible. As described in Section 2.6, Puschner and Schedl [1997] show how for a control flow graph the WCET can be calculated by using the IPET approach. The ISPTAP tool also employs the IPET approach. For the generation of the ILP formulation it uses the VIVU translated supergraph enriched by loop bounds, execution cost, and memory penalty. To find the maximal value of the objective function the ILP solving tool `lp_solve`[4] is used [Berkelaar et al., 2008]. It is also employed to find the assignments for the static memories.

5.2.6 Validation of the CarCore Timing Model

This section provides a short validation of the CarCore timing model implemented by the ISPTAP for several small benchmarks. Therefore, a set of 10 micro-benchmarks from the Mälardalen benchmark suite [Gustafsson et al., 2010] was selected. A short overview of the characteristics of the benchmarks is given in Table 5.2. The characterisation is partly obtained from [Mälardalen Real-Time Research Center (MRTC)].

These benchmarks were executed by the cycle-accurate SystemC simulator of the CarCore processor (refer also to Section 6.3.1) without any on-chip instruction memory. The memory access time for instruction and data memory is the same and is assumed as 4 cycles. After simulation the executed path of each benchmark was extracted and fed into the ISPTAP tool by creating adequate flow constraints. Thus the analysis performed by ISPTAP estimates the cost in cycles of the same path that was also executed by the CarCore simulator. The results of the analysis by the ISPTAP tool are provided in Table 5.3. The third column of the table shows the differences of the estimates to the number of cycles needed to execute the benchmark on the CarCore processor simulator. This difference quantifies the overestimation of the analysis. As the shown in the table the overestimation ranges from very small values, like for Fdct with 2.8%, to a rather large impreciseness, as for Duff with 31.7%. Because the overestimation is a relative measure that depends on the cycle count needed to execute the considered path, the total estimate calculated by ISPTAP is shown in the second column the table. Furthermore, the number of basic blocks is provided to give insight in the length of the analysed path.

[4]The `lp_solve` version 5.5.0.13 was used.

Table 5.2: Benchmarks for validation of the CarCore timing model (* including benchmark function and depending on the benchmark a harness and/or an initialisation function)

Benchmark	**LOC**	**Structured Code**	**Loops**	**Nested Loops**	**Indirect Jumps**	**Function Count***
Adpcm	879	✓	✓	–	–	17
Bsort100	128	✓	✓	✓	–	3
Cover	240	✓	✓	–	✓	4
Crc	128	✓	✓	–	–	3
Duff	86	–	✓	–	✓	3
Fdct	239	✓	✓	–	–	2
Fibcall	72	✓	✓	–	–	2
Fir	276	✓	✓	✓	–	2
Matmult	163	✓	✓	✓	–	6
Nsichneu	4,253	✓	✓	–	–	1

Table 5.3: Overestimation of the CarCore timing by ISPTAP (The estimated execution time is compared to the execution time determined with the CarCore simulator for the same path.)

Benchmark	**Estimate (in Cycles)**	**Overestimation**	**Basic Blocks on Path**
Adpcm	1,057,720	19.3%	26,326
Bsort100	7,423	7.7%	405
Cover	8,510	16.1%	670
Crc	124,901	24.6%	6,406
Duff	3,050	31.7%	152
Fdct	3,531	2.8%	21
Fibcall	912	13.0%	21
Fir	5,422	9.3%	275
Matmult	428,108	10.8%	11,334
Nsichneu	12,394	4.7%	1,252

In average the overestimation of ISPTAP is about 14% for the CarCore processor without any on-chip memory (i.e. off-chip memory only). The timing analysis used the very same path as was executed by the CarCore SystemC simulator for each benchmark. Therefore, the overestimation cannot be caused by differences in the path assumed by ISPTAP and executed by the simulator.

Landet modelled the superscalar pipeline of the CarCore processor using the OTAWA framework (see Section 2.6.3) and shows in [Landet, 2008; Landet, 2009] results for the overestimation for a subset[5] of the Mälardalen benchmarks that are in average slightly below 4%. Thus by comparing the benchmarks used by Landet and the set of benchmarks used in Table 5.3 the overestimation of ISPTAP is about 3 times higher than the results from [Landet, 2008][6].

The reason why the overestimation for ISPTAP is higher depends mainly on the simpler pipeline timing model, that takes no interactions between different basic blocks into account, i.e. the timing of every basic block is considered as independent of any previous or subsequent basic blocks. Thus to prevent underestimating this effect, the timing model of the ISPTAP is more pessimistic, which leads to higher overestimations. In [Landet, 2008] the pipeline is

[5]The following benchmarks analysed here are also used by Landet: Bsort100, Fdct, Fibcall, Fir, and Nsichneu.

[6]Unfortunately, the benchmarks that result in an overestimation of above 15% for ISPTAP were not part of the evaluation of Landet [2008].

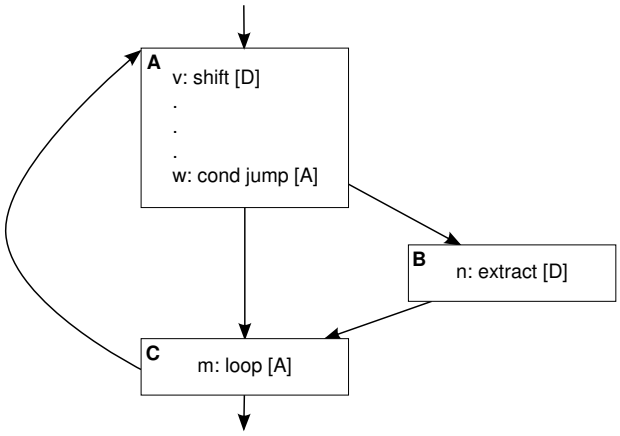


Figure 5.5: Two basic blocks from Crc benchmark, showing inter-basic-block timing effects

modelled using *parametrized execution graphs*. This allows to take the effect on the timing of a certain basic block caused by basic blocks that are executed before and after into account. In [Rochange and Sainrat, 2009], which described the usage of parametrised execution graphs to reduce the overestimation for basic blocks in out-of-order processors, it is stated that the execution timing of a basic block can increased or reduced by prefix/prologue and suffix/epilogue basic blocks/instructions. This is even possible for in-order superscalar processors, as the CarCore processor. The usage of execution graphs for pipeline analysis was introduced by [Li et al., 2004] and were extended to superscalar pipelines and integrated to OTAWA by [Barre et al., 2006; Rochange and Sainrat, 2009]. The timing of the instructions in the processor's pipeline can also be modelled precisely by the use of abstract interpretation, as Thesing [2004] shows. In [Engblom, 2002] the *timing effects* of dynamically scheduled pipelines are also discussed. Nevertheless, all these approaches will deliver a tighter timing of the processor pipeline, if the simple pipeline timing model of ISPTAP would be extended. For the discussion of the pipeline analysis and more related work refer to Section 2.6.

In the following an example is given that shows how the omitting of the execution context of basic blocks effects the preciseness of the ISPTAP's WCET estimates.

Example. The effect of the influence on the timing of a basic block by a previously executed basic block is shown in Figure 5.5. The figure shows three basic blocks (A, B, and C) from the Crc benchmark of the Mälardalen suite. Depending on a condition the conditional jump w either proceeds to block B or jumps to block C. If the basic block B is executed, its single data pipeline instruction (n) is followed by the address instruction m which is in this case from basic block C. Due to the architecture of the CarCore it is possible that both instructions can be executed within one cycle (refer to Table 5.1). By the parallel execution of instructions n and m the basic block B is executed at the same time as the basic block C. So the execution cost of basic block B and C is 3 cycles (one cycle for the jump instruction m and 2 cycles as branch delay). If both basic blocks are analysed independently from each other, 4 cycles are estimated for their execution cost. Because the timing model of ISPTAP is not capable of considering the execution context of a basic block when calculating its cost, the extra cycle is charged by ISPTAP for the basic block cost of both blocks. Because the blocks A, B, and C form a loop, which is executed rather often in the Crc benchmark, the overestimation for the execution cost of basic blocks B and C causes ISPTAP to overestimate the WCET of the entire benchmark by 4%. ♦

Another timing effect that is beyond the scope of one basic block is caused by the initial content of the instruction window (IW) on begin of the execution of a basic block. The content of the IW is crucial for a tight estimation of a basic blocks execution cost, because if an instruction is not contained the time to fetch the instruction is to be charged. Usually the IW already contains some instructions of the basic block on begin of its execution, if these instructions were fetched during the execution of the predecessor, i.e. the basic block is activated by continuous addressing. But if a basic block is entered by a jump, the IW was flushed and is empty on execution start of the basic block. So the distinguishing of the execution context of a certain basic block is important for its execution cost. The ISPTAP tool can only distinguish, if a basic block is always entered by continuous addressing or not. In the first case it assumes the IW content is available that was left by its predecessor. In the second case ISPTAP timing model has to assume that the IW is empty, because it cannot distinguish the type of activation of the basic block, since this is dependent on the prior execution context (e.g. the basic block may entered by a jump or by continuous addressing, which is not distinguished by ISPTAP). So it is assumed that those basic blocks are always entered by a jump, resulting in an empty IW on the begin of their execution. This causes an overestimation, because the IW content has to be provided by fetches, which are possibly not necessary. Anyhow, ISPTAP delivers for each basic block a safe upper bound of its execution cost.

Since the focus of the ISPTAP tool is on the analysis and comparison of the WCET impact of different instruction memories the level of overestimation shown in this section is considered as acceptable. To improve the results of the ISPTAP tool it is possible to tighten the pipeline timing model by employing one of the analysis techniques discussed above. For example execution graphs can be used to reduce the overestimation by taking previous basic blocks into account [Rochange and Sainrat, 2009]. This was already done for the CarCore processor in the work of Landet [2008]. To improve the pipeline timing model the pipeline execution cost analysis that is performed for each basic block separately has to be enhanced to model also inter-basic-block timing effects. By the structure of ISPTAP the additional software development effort to integrate another pipeline timing model is limited.

A further cause for overestimation is that context dependent flow constraints like loop bounds cannot be modelled in ISPTAP. Because no such content dependent flows occur in the analysed benchmarks above, no impreciseness can be charged to the flow constraints. However, ISPTAP supports only static flow information and thus only context independent loop bounds. So for loops with a variable maximum iteration count that depend on the context of the loop the maximum possible loop bound has to be considered for all contexts. For example for a nested loop in which the iteration count of the inner loop depends on the iteration of the outer loop ISPTAP has to use the maximum possible iteration count for every execution context of the inner loop. Thus the execution cost of the nested loop will be overestimated. To support context dependent flow constraints the capabilities of ISPTAP have to be extended to import flow facts for different execution contexts and assign them to the correct position in the supergraph and to the flow variable in the IPET formulation of the application. The issues of the formulation of proper context dependent flow information and the mapping of these information to the flow constraints in an IPET formulation are addressed by Engblom and Ermedahl [2000]. Therefore, Engblom and Ermedahl introduce the concept of *scoped flow information* that allows to model a wide range of flow descriptions. An implementation of this concept, or any similar that is as powerful, requires changes in ISPTAP that are restricted to the processing of the flow facts and the enrichment of the graph that is used to build the ILP with the flow information. These enhancements are also left for future work.

5.3 Application Requirements for the D–ISP

The D–ISP raises some restrictions to the style of the application code that have to be respected, otherwise the D–ISP cannot be used for the application. These restrictions mainly affect compiler optimisations, but also can influence decisions of the programmer. In the following all restrictions and major code guidelines to use the D–ISP are discussed.

5.3.1 Application Programming and System Guidelines

As usual in software development for embedded (real-time) systems the programmer has to know the architectural design parameters of the system before developing the application. Using the D–ISP requires to respect the three following design parameters of the D–ISP:

- D–ISP size (and its block size)
- mapping & lookup table size
- context stack memory size

The size of the D–ISP has to be known by the programmer to ensure that every function (that is intended to be maintained by the D–ISP) fits the D–ISP. If a function is identified that cannot be loaded into the D–ISP, the function needs to be split. Otherwise it can be executed only without using the D–ISP. The check of the function sizes can be done automatically, but the splitting of outsized functions has to be performed by the programmer, since the splitting of a function may strongly affect the timing of the application.

Beside the size of the D–ISP the maximum function size is also delimited by the instrumentation, if the FLE instruction is used. As shown in Section 5.1 the maximal size that can be encoded is 64 KiB. Functions larger than this size have to be split by the programmer. Otherwise the FDBP instrumentation can be used or the decoding of the FLE instruction can be altered.

To know the size of the mapping and lookup table is not necessary, but can help the programmer to reduce function evictions caused by mapping/lookup table overrun. This overrun can only happen, if in the same time more functions are maintained by the D–ISP than table entries exists. The usual case of evicting a function is when the available D–ISP memory is too small on function load. But if more functions fit the D–ISP than it can maintain in the mapping/lookup table, a function also has to be evicted by table overrun. This is more likely to happen, if the ratio of the scratchpad size and the mapping/lookup table entries ($\frac{size_{D-ISP}}{no_{func}}$) is rather high. In case of the no_{func} smallest functions are larger than the scratchpad size, the function eviction due to table overrun is not possible. To be sure that such overrun will not happen the programmer can check the call graph of the application. Notice that such overrun is not critical for the correctness of the application, because the table overrun is handled exactly like the function eviction on D–ISP memory shortage.

The D–ISP has a special context stack to identify the caller function on return. Since the size of this memory is limited, the programmer can not build an application with a call depth that exceeds the capacity of this memory. So it has to be ensured that the number of entries in the context stack is suitable for the application. This can be verified by determining the longest path in the application's call graph including the maximum depth of recursive calls. Usually processors also limit the call depth, e.g. the TriCore features *Context Save Areas* (CSAs) [TriCore, 2007a, p. 4-3] in which the contexts are saved e.g. on function call. The number of CSAs is limited to 2^{16}. Using this size for the context stack memory is not suitable, because it would use 256 KiB (by using Equation (3.9) for calculation of size(CSM) and an address width for native addresses

of 32 bit) of on-chip memory for holding the context stack only. Therefore, a smaller maximum depth needs to be selected. To control this the TriCore allows to use a *Call Depth Counter* (CDC) [TriCore, 2007a, p. 3-6] to monitor the number of used contexts. It can be used to execute a trap that disables the D–ISP on pending context stack memory overflow by e.g. unintended recursion. Then the application can be correctly executed without the D–ISP until the D–ISP may be enabled again on return to a maintained context. Due to this recursion is supported by the D–ISP, if the recursion depth is limited and manageable by the context stack memory. Notice that from the D–ISP content point of view recursion is not an issue, since every function is contained at most once in the D–ISP.

5.3.2 Code Style Restrictions

Beside the requirements that need to be respected by the application the D–ISP prohibits some code optimisations by the compiler.

To allow loading of a function correctly into the D–ISP the code of the function has to be a closed representation with the first instruction of the function at the lowest address. Therefore, aggressive block reordering and scheduling optimisations by the compiler should be disabled to safely support the D–ISP. Because such optimisation can lead to a code representation that does not allow to load the complete function from beginning to end into the D–ISP by using the FDBP or the FLE method, the D–ISP may not operate properly when these optimisations are used.

Furthermore, any optimisations that break the process of function activation by call or that leave the code of a function without a return cannot be used. An example for this is the *tail-call optimisation* [Muchnick, 1997; TriCore, 2003] in which a call to a function is replaced by a jump to another function. Since the activated function is entered without a call, the D–ISP controller is not triggered to activate it. Thus the code of this function cannot be fetched from the D–ISP, no matter if it is present in the D–ISP or not. However, the FMUX (refer to Section 3.3.2) ensures that fetch requests to the function entered by jump are routed and handled by the off-chip memory controller. Notice that *tail-recursion elimination* is supported by the D–ISP, since in this optimisation the function activates only itself by jumping. So the recursively activated function is still the function that is active in the D–ISP controller.

The inlining of functions (including *in-line expansion* [Muchnick, 1997] on assembly-language level) is supported by the D–ISP, if the instrumentation uses the function sizes including the inlined code. Because the instrumentation tools proposed in Section 5.1 use the assembly code or the linked executable for instrumenting the function size information, it is ensured that the correct size for the functions including any inlined code is used.

To current knowledge any other compiler optimisation that was examined has no negative impact on the D–ISP. All applications that use the D–ISP are build with the HighTec TriCore GCC 3.3 [HighTec, 2003] using the following configuration, that adds debugging symbols, uses the optimisation level 2, uses the TriCore 1.3 instruction set, enables all code generation erratas, disallows basic block reordering, and aligns every function to 64 bit addresses (to support an easy determination of the FLE instruction by the D–ISP controller):

```
-g -O2 -mtc13 -mall-errata -fno-reorder-blocks -falign-functions=8
```

5.3.3 D–ISP Control Interface

On system start up the D–ISP is deactivated by default, because the initialisation code that is executed only once does not need to be buffered in the D–ISP. Further, the application initialisation phase is usually not timing critical and from memory analysis point of view a clean

D–ISP content is favourable on application start, refer to the discussion of sensitivity of the replacement policies on an unknown state in Section 4.3.4. To activate the D–ISP the D–ISP content management is controllable by the processor. Therefore, a set of CSFRs is assigned to the D–ISP. These core special function registers can be written by the MTCR instruction (shown in Figure 5.1), that is also used to implement the FLE instruction. The register space assigned to the D–ISP starts at address `0x6000` and contains four registers. The used core registers do not conflict with the registers used by TriCore processor [TriCore, 2007a, pp. 14-1ff]. Thus the processor is capable to control the D–ISP by the following four commands:

- **Start At Address**: Starts the operation of the D–ISP with a call of a selected function.
- **Flush**: Deletes the content of the following management structures: lookup table, context stack memory and context register. After the flush the D–ISP is in an empty state.
- **Deactivate**: Pauses the operation of the D–ISP. All instructions will be fetched from the off-chip memory and the D–ISP is not sensitive to call or return instructions.
- **Reactivate**: Reactivates the D–ISP with the same content of memory and management structures as it was deactivated. The D–ISP has to be reactivated at the same execution context as it was deactivated, otherwise its behaviour is not specified.

By using the **Start At Address** command the application enables the D–ISP and it is used until application end or D–ISP deactivation.

The **Flush** command completely invalidates the D–ISP content including all maintenance memories and the content of the scratchpad. This is ensured by simply dropping the content of the lookup table, the context stack memory and the context register, such that the content of the scratchpad can no longer be accessed. By this also the mapping table entries are indirectly invalidated, because the lookup fails for every function, independently of the mapping table's content. The time to perform the flush depends on the lookup table size and the size of the context stack memory. Due to the fact that both memories can be accessed in parallel the flush takes the maximum of the access times for accessing all entries in the lookup table and in the context memory:

$$cycles_{flush} = \max\left(\frac{no_{func}}{no_{look}},\ no_{stack\_depth}\right)$$

Since the lookup table can access no_{look} entries in parallel, a higher lookup width does not only speed up the function lookup (as shown in Equation (3.7)) also the time to flush the content is reduced. Notice that the latency of the flush does not depend on the size of the D–ISP memory. The processor is not stalled on execution of the flush. Therefore, the D–ISP is inaccessible to the processor for any function activation or command until the tables are not completely cleared.

The **De-/Reactivation** of the D–ISP can be used, if a certain code region does not need to be executed using the D–ISP, as for example maintenance code that is not time-critical. Beside that the D–ISP can be deactivated, if the execution of a code part may put the D–ISP into an unknown or undesirable state. This is e.g. the case for handling interrupts: if during interrupt handling the D–ISP is deactivated, the interrupt handling is fully transparent to the D–ISP content and thus does not need to be considered on memory content analysis. Another use for the de-/reactivation is the execution a software scheduler or D–ISP incompatible legacy code.

5.3.4 Legacy Code Compatibility

If a system equipped with the D–ISP has to execute legacy code, i.e. code that is not instrumented and that may have an unknown structure, the D–ISP can only be used with restrictions. In the

following two possibilities are described that enable the execution of legacy code in a system with a D–ISP as first level instruction memory.

The first option is to deactivate the D–ISP before executing uninstrumented functions and reactivating it on their return. This requires the application programmer to surround the calls to legacy code with D–ISP de-/reactivate commands.

The second option is to use the D–ISP implementation that determines the function size by the FLE instruction, because every function that is not instrumented will be ignored by the D–ISP controller inherently. On return from a function the D–ISP is reactivated, if the caller function used the D–ISP before. Therefore, the D–ISP maintains the context stack also for non-maintained functions. Notice within the uninstrumented functions no code style restrictions need to be followed, unless the context stack of the D–ISP is not broken. So no additional effort has to be invested to use the D–ISP for legacy code, when the FLE instruction is used by the D–ISP controller to determine the function size.

Chapter 6

Evaluation

This chapter provides results of the hardware implementation of the D–ISP in terms of the logic complexity of the D–ISP controller and the memory overhead induced by the D–ISP's helper memories. The discussion of the needed hardware amount is important to rate the cost of implementing the novel dynamic instruction scratchpad design in current embedded systems. This facet of the D–ISP is highlighted in the Section 6.1.

The major part of the evaluation presented in this chapter is the quantification of the impact of the D–ISP on the WCET estimate of the system. The WCET estimate is the important metric that is to be considered in hard real-time systems. Hence, the D–ISP is designed to ease its analysis and provide lower WCET estimates than other common instruction memories. Therefore, the evaluation in Section 6.2 determines if the additional hardware effort of the D–ISP pays off in lower WCET estimates. To obtain the WCET estimates the ISPTAP analysis tool that was presented in Section 5.2 is used. The evaluation will show that the D–ISP is able to outperform common memories used in embedded real-time systems like instruction caches and scratchpads with fixed content.

Estimates regarding the average performance of the D–ISP are provided in Section 6.3 of this chapter. The investigation of the average case performance gives insights how a system with D–ISP behaves usually. Since the WCET is the metric of interest a good average case performance is not of major importance, but it is worth looking at the performance of the common case.

The Section 6.4 briefly discusses the complexity of the D–ISP analysis that was proposed in Section 4.3. It will be shown that the maintenance of all possible D–ISP memory states during the memory content analysis poses a manageable degree of analysis complexity for small benchmarks, but does not scale for large applications.

6.1 Hardware Effort Estimation

The Section 3.3 showed how the D–ISP can be implemented in hardware. In this section the cost of this implementation is evaluated. Furthermore, another main subject of this section is the cost of the predictability for an instruction memory, i.e. the quantification of the additional cost of the D–ISP comparing to a common first level instruction memory. Therefore, the D–ISP is compared to a traditional instruction cache.

The hardware effort of the D–ISP is partitioned into three facets: logic amount, memory overhead, and timing characteristics. The logic amount denotes the complexity of the hardware implementation of the D–ISP controller. The memory overhead of the D–ISP represents the

Table 6.1: Features of Altera Stratix II EP2S180F1020C3 according to [Altera, 2009]

Feature	Value
ALMs	71,760
ALUTs	143,520
M512 RAM blocks	930
M4K RAM blocks	768
M-RAM blocks	9
Total RAM bits	9,383,040

additional memory needed to maintain the content of the D–ISP, namely this is the size of the helper memories. The timing characteristics discussed for the D–ISP implementation basically address the maximum possible frequency of the D–ISP controller and the timing of the fetch control process of D–ISP. Before discussing the hardware effort of a prototypic D–ISP implementation the evaluation methodology is described. Parts of the following section were already published in [Metzlaff et al., 2011a].

6.1.1 Evaluation Methodology

To estimate the hardware effort of the design the D–ISP was implemented in a Field-Programmable Gate Array (FPGA) using the hardware description language VHDL[1]. The hardware effort of a design using an FPGA corresponds to the hardware complexity of the design when implementing it in other technologies, for example in structured-ASICs[2] as pointed out by Hutton et al. [2006]. Kuon and Rose [2007] quantify the gap between an FPGA and ASIC implementation in terms of area, timing, and power consumption. For different designs that using only logic and registers the FPGA requires in average about 35 times more area than an implementation in an ASIC. This difference is in reduced in average to 33 for designs that also use memory blocks. A study of Wong et al. [2011] compares the area and the delay of processor building blocks when implementing them in FPGA and in custom CMOS. It is discussed that especially high associative caches cannot be implemented in FPGAs efficiently. This is due to the costly representation of CAMs[3] in FPGAs.

For all evaluations in this section the Altera Stratix II EP2S180F1020C3 FPGA was used. The main features of this specific FGPA are listed in Table 6.1. The Stratix II FPGA consists of an two-dimensional array of logic, memory, and special digital signal processing blocks [Altera, 2009]. The latter is not of interest for the D–ISP implementation and thus the digital signal processing blocks are omitted in the further description. All block are connected by row and column interconnections. Each logic array block (LAB) consists of eight adaptive logic modules (ALM) that contain two adaptive lookup tables (ALUTs) each. The ALUT is the basic cell for the logic synthesis of the design by the vendor's tool chain. Therefore, the ALUT is used for the further discussion of the logic amount estimation. Despite that, the ALM is the basic building block for the implementation of the logic in the Altera Stratix II FPGA [Altera, 2009].

An ALM consists of a lookup table resource, two registers, two full adders and a carry, shared arithmetic, and register chain in- and output. The structural overview of an ALM in shown in Figure 6.1. The lookup table resource of the ALM can be split into two ALUTs with four inputs, such that one ALM can implement two functions with for up to four inputs each. Furthermore,

[1] VHDL – very-high-speed integrated circuits (VHSIC) hardware description language
[2] ASIC – application-specific integrated circuit
[3] CAM – Content Addressable Memory, refer e.g. to [Pagiamtzis and Sheikholeslami, 2006]

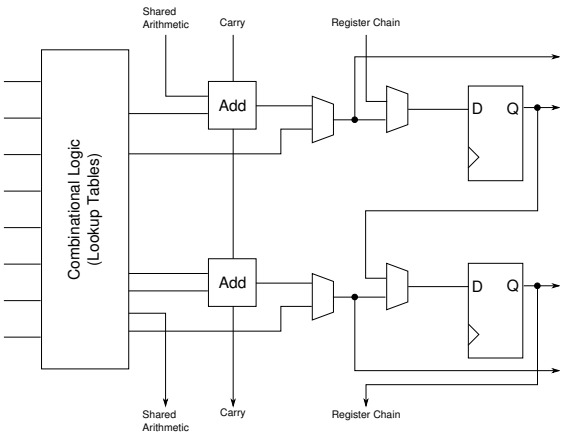


Figure 6.1: Adaptive Logic Module (ALM) of Altera Stratix II as depicted in [Altera, 2009]

one ALM can also implement any six-input function and some seven-input functions. For detailed information on the ALM structure see [Altera, 2009]. The two registers in the ALM allow storing input values or results from the lookup table or the adders. Synchronous processes in the design are implemented using these registers as latches. It is also possible to connect multiple ALMs bypassing the registers and so creating more complex asynchronous logical functions.

The memory blocks in Stratix II FPGAs are dual-ported memories with different port width and capacities: the M512 RAM (with 512 bit), the M4K RAM (with 4 Kibit), and the M-RAM (with 512 Kibit). This block memory can be used to model any internal storage of the design such as the register set of a processor, cache memory (including tag memory), scratchpad memory, and the D–ISP memory including the helper memories. It is also possible to use the registers of the ALMs to implement internal storage, but in this case the logic amount is increased and precious ALMs are wasted.

The D–ISP design and the CarCore host processor integration was tested using the Stratix II DSP Development Board [Altera, 2006]. For design synthesis and timing analysis the Altera Quartus II 9.0 tool chain was used. The Altera Stratix II EP2S180F1020C3 FPGA provides much more resources that any design that is evaluated in the following. This is of importance since, if the logic amount needed by the design is close to the limits of the FPGA, placement and routing problems may occur during synthesis that blur the necessary logic amount of the design.

6.1.2 Hardware Complexity

The logic amount of a design implemented in an FPGA can be split into the used number of ALUTs and registers. Also the number of DSP elements can be considered, but since no multiplication or signal processing is performed in the designs that are synthesized such elements will not be used. The number of ALUTs that a design uses provides information about the complexity of the internal logic, whereas the register use allows to draw conclusions about the amount of data that is to be stored between the clock cycles.

To allow a better quantification of the cost of the D–ISP the logic usage of the instruction caches of two common freely accessible processors are provided for comparison. First the hardware effort of the instruction cache of the OpenRISC processor [OpenCores, 2006] is quantified. The second processor for which the instruction cache implementation is briefly investigated is

Table 6.2: Resource usage of the OpenRISC instruction cache for the Stratix II EP2S180F1020C3

Size	ALUTs	Registers	Block Memory
512 B	211	180	608 B
4 KiB	207	186	4,768 B
8 KiB	203	188	9,472 B

the Leon3 [Aeroflex Gaisler AB] processor. Both are also synthesised with Altera Quartus II 9.0 for the Altera Stratix II EP2S180F1020C3 FPGA.

OpenRISC

The OpenRISC 1000[4] is a 32/64 bit load/store RISC architecture with DSP, vector, and floating point extensions [OpenCores, 2006]. The architecture is designed for networking and embedded system platforms. The OpenRISC 1200 [Lampret, 2011] implements the architecture and is available as a synthesiseable Verilog design under LGPL[5]. The design features instruction and data caches, FPU, MMU, interrupt controller, timer, and power management. To build a SoC using the OpenRISC 1200 it is equipped with a Wishbone bus interface, that allows the connection to other IP cores. The OpenRISC 1200 currently supports only direct mapped caches with varying sizes from 512 B to 32 KiB [Lampret, 2011]. The cache lines have a width of 16 B, except for the 32 KiB configuration that uses 32 B per line.

The logic amount used of the instruction cache subsystem of the OpenRISC 1200 for the three different cache sizes is shown in Table 6.2. As the table shows the logic amount of the instruction cache subsystem is basically independent of the size of the cache. This caused by fact that in the implementation only the size of the underlying tag and cache memory is altered and the tag length is adapted to the cache size. The small differences in the ALUT count may be caused by layout changes of the designs in the FPGA.

Leon3

The Leon3 processor [Aeroflex Gaisler AB] is an IP core based on the 32 bit SPARC V8 architecture. It features a 7-stage pipeline, multiprocessor configurations (including support for cache coherency), fault-tolerance, FPU, MMU, hardware support for MUL, DIV, and MAC (multiply accumulate) instructions, interrupt control, power management, and debug support. The Leon3 processor supports beside the direct mapped instruction cache a cache associativity of up to 4 ways and the LRU, LRR, or random replacement policy [Aeroflex Gaisler AB, 2010]. The Leon3 is available[6] as source code under GNU GPL license. For building SoCs an IP core library is provided. Additionally the Leon3 is equipped with an AMBA AHB (Advanced High-performance Bus) interface to connect custom IP cores.

To determine the hardware cost of the Leon3 instruction cache subsystem a minimal configuration is created and then the additional cost for the different cache configurations is extracted[7]. The baseline Leon3 configuration does not support any complex instructions and has no optional features (as FPU) and no caches. The baseline processor uses 4,546 ALUTs and 3,182 registers. Also it uses 2,048 B of block memory e.g. for the processors register set. In Table 6.3 the number

[4] see `www.opencores.org/openrisc` [Online, last accessed on 3rd March 2012]

[5] LGPL – GNU Lesser General Public License

[6] see `www.gaisler.com` [Online, last accessed on 3rd March 2012]

[7] The cache subsystem was not independently synthesised, because the cache memories and the controller are not connected by a separated entity as for the OpenRISC. The cache components are interweaved in the processor structure and a separation would require changes in the implementation of the Leon3 architecture.

Table 6.3: Resource usage of the Leon3 instruction cache for the Stratix II EP2S180F1020C3

Associativity	Replacement Policy	Size per Set	ALUTs	Registers	Block Memory
1	direct mapped	1 KiB	188	12	1,144 B
		2 KiB	185	13	2,280 B
		4 KiB	190	14	4,544 B
2	LRU	1 KiB	342	87	2,288 B
		2 KiB	435	121	4,560 B
		4 KiB	517	187	9,088 B
4	LRU	1 KiB	541	274	4,576 B
		2 KiB	642	445	9,120 B
		4 KiB	973	767	18,176 B

of ALUTs and registers that are additionally needed for the different instruction cache configurations are shown. Configurations from 1 to 4 cache sets with direct mapped and LRU replacement policy are chosen. The size of each cache set varies between the minimal size of 1 KiB and the size of 4 KiB. The cache line size of every instruction cache configuration is 32 B.

It can be seen in the Table 6.3 that the LRU replacement policy needs significantly more ALUTs and registers than the direct mapped replacement strategy. This is because of the more complex lookup mechanism that requires a tag comparison per cache set [Jacob et al., 2007] and the maintenance of access counters per set for each cache line (to determine the least recently used cache line on a cache miss). Therefore, the hardware effort also depends on the size of the LRU cache, which is shown in the table for cache set sizes of 1 KiB, 2 KiB, and 4 KiB.

Notice that the overall block memory is nearly independent of the applied replacement policy, e.g. for a size per set of 1 KiB the block memory of the 2-way set associative LRU cache is twice the memory used for the direct mapped cache. This shows that the additional status information needed by the LRU replacement policy is hold in logic registers. That explains the higher number of used registers for larger cache sizes and higher associativities.

The main source of complexity regarding the used logic cells is the associativity of the cache, e.g. the 2-way 4 KiB per set LRU cache needs less ALUTs and registers than the 4-way 1 KiB per set LRU cache and has twice the capacity. This is caused by the lookup logic that has to select the correct cache set within one cycle during hit detection on every cache access. Also the information of each cache line per set that was selected by the tag part of the address has to be updated for each cache hit. According to Milenkovic et al. [2003], $n_{ways} \cdot \log_2(n_{ways})$ status bits are required per cache set to maintain the access history of an LRU cache, with n_{ways} as the number of cache lines per set. So with a higher associativity of the cache the complexity to update the data, which is needed by the replacement policy to select the least recently used cache line on miss, increases.

D–ISP

To evaluate the hardware complexity of the D–ISP controller it was implemented in VHDL. In this section the D–ISP with FIFO replacement policy will be considered. The implementation of the stack-based replacement policy is discussed separately in Section 6.1.5. The D–ISP controller is implemented using the two processes for *fetch control* and *content management* and the helper memories (for the *lookup table*, the *mapping table*, and the *context stack*) as depicted already in Figure 3.3 (p. 56) of Section 3.2.3. For the helper memories the block RAM of the FPGA is used. The context register that manages the communication between the two processes is implemented

Table 6.4: Resource usage of the D–ISP controller for the Stratix II EP2S180F1020C3

Lookup Width no_{look} (in Functions)	**ALUTs**	**Registers**
2	806	608
4	**809**	**613**
8	986	653
16	1,159	658
32	1,505	665
64	2,185	672
128	3,512	677
256	6,218	682

using registers. The *content management* process of the D–ISP controller is connected to the helper memories as depicted in Figure 3.11 (p. 72). For the discussion of the hardware effort of the D–ISP controller the synchronous *fetch control* as shown in Figure 3.10(a) is considered. For further implementation details see Section 3.3. The implemented D–ISP controller supports the FLE function size instrumentation, as described in Section 3.3.4.

The Table 6.4 shows the hardware effort of the D–ISP controller without the helper memories needed to store and maintain the content of the D–ISP in numbers of ALUTs and registers. Anyhow, the connection ports to the block memories that contain the helper memories are included. In the table the hardware effort of D–ISP controller with different lookup widths (no_{look}, refer to Section 3.3.4) is presented. The lookup width represents the number of entries that can be checked from the lookup table within one cycle during hit detection. It can be seen that with increasing lookup width the hardware effort increases significantly, especially in terms of ALUTs. This is not surprising, because the lookup width determines the number of parallel comparisons that have to be done within one cycle to detect if a function is a miss or a hit. Since the parallel comparisons are very costly in hardware[8], the lookup width no_{look} dominates the logic cost of the D–ISP controller for higher no_{look} values in terms of ALUTs. So for a lookup width of 64 functions about 2.7 times the number of ALUTs is needed to implement the D–ISP controller than for the baseline version with a lookup width of 4. Because any larger lookup width results in a significantly higher hardware cost, it is suggested to use at most 4 functions to be compared within one cycle. To support more than 4 functions the multi-cycle lookup can be used, which was described in Section 3.3.4. For example if the D–ISP supports 32 functions and the lookup width is kept at 4 ($no_{func} = 32$ and $no_{look} = 4$), the function lookup will take 8 cycles at maximum (refer to Equations (3.10) and (3.11)). The evaluation of the function lookup width in Section 6.2.7 also shows that a lookup width of 4 is optimal in terms of impact on the WCET estimates and required hardware complexity.

The Table 6.4 also shows the used registers. It can be seen that the number of used registers is quite high, but it only slightly increases for larger values of no_{look}. The high amount of registers is needed, because the two D–ISP controller processes need numerous input and output signals to be connected with the CarCore pipeline, the memory controller, the FMUX, and the helper memories. Furthermore, the *context register* is implemented using registers and the *content manager* needs to maintain information in logic registers, e.g. the state of the finite state machine, the write and eviction pointers, the number of blocks to fetch until the whole function

[8]For example [Hill, 1988] shows the implementation differences of direct mapped and set associative caches in terms of hardware complexity and access time. Hill states that the associative lookup and comparison increases the cost of the hardware implementation significantly.

is loaded, and the pointers to the currently active entry of the context stack. The number of necessary registers increases only slightly when the lookup table width is increased, because only the port width of the connection to the lookup table is increased (and thus a few additional registers are needed for the additional input signals). Beside that within the *content management* only the processing of the hit detection that is done within a cycle is altered. Therefore, no extra registers are needed for the hit detection, if the lookup width is increased.

When comparing the logic complexity of instruction caches to the D–ISP presented in the Tables 6.2, 6.3, and 6.4 the D–ISP requires clearly more hardware resources to be implemented. So the largest 4-way set associative cache requires a hardware amount that is in the same order of magnitude as a D–ISP with a lookup width of 8 functions. Furthermore, it can be seen in Table 6.3 that with increasing associativity the needed ALUTs and registers significantly increase. This is due to the higher implementation effort that is needed for the more complex cache set selection logic and the tag comparison [Hill, 1988]. The sensitivity of the hardware complexity on increased associativity is similar for the D–ISP. In the D–ISP also larger lookup widths require a higher number of parallel comparisons and the use of a wider multiplexer to select the matching function table entry.

In contrast to the cache the D–ISP's hardware complexity is not dependent on the size of the memory. This is because the management tables are located in on-chip SRAM memories instead of implementing them in logic registers. The necessary amount of additional memory that is given by the D–ISP management tables is discussed in the Section 6.1.3.

In the following the D–ISP is compared to the method cache of Schoeberl (refer to Section 2.3.1), which is very small due to the usage of a cache structure [Schoeberl, 2004]. In [Pitter and Schoeberl, 2010][9] the size of a single core JOP including an 2 KiB method cache is 3,329 logic elements (LEs) synthesised with Altera Quartus II for an Altera EP2C35 FPGA [Altera, 2008b]. To allow a rating of the hardware amount required by the method cache and the D–ISP the D–ISP with a lookup width of 4 functions was synthesised for the same FPGA resulting in an LE count of 1,412[10].

Therefore, it can be assumed that the D–ISP is multiple times larger than the method cache of the JOP. This is because the method cache content is mainly controlled via two instructions within invoke and return microcodes: The instruction `ldbcstart` that pushes the address of the activated method onto the stack and `stbcrd` that initiates a DMA transfer [Schoeberl, 2005]. The DMA transfer loads the complete method into the cache by requesting all cache lines from the memory. Hence, for the method cache itself an ordinary instruction cache can be used. By a fully associative FIFO replacement policy for the conventional instruction cache the replacement policy of the method cache, which is on the granularity of methods, is ensured. So the use of a conventional cache and the cache filling via DMA transfers poses a low hardware complexity. The D–ISP is more complex, because it has to decode the FLE, manage the address space of the scratchpad memory, trigger the function load, set the write address for the incoming memory blocks, set the current context, and maintain the helper memories. The prototypic implementation of the D–ISP leaves potential to reduce its complexity by a tighter integration into the host processor and by the tailoring of the internal state machine to a fixed off-chip memory latency. Anyhow, it is not assumed that it reaches the low hardware amount that the method cache requires.

[9] The paper discusses a multicore version of the JOP, so the hardware estimation also contains modules that are not necessarily needed in a single core JOP: a bus interface [Schoeberl, 2007], a memory arbiter, and a synchronisation unit [Pitter and Schoeberl, 2007].

[10] In [Schoeberl, 2008] the size of the JOP including a 1 KiB method cache is 1,077 LC (logic cells) on an Altera EP1C6 FPGA [Altera, 2008a]. The D–ISP requires 1,461 LE for the same FPGA. But since the conversion of LC to LE is ambiguous, these values cannot be directly compared.

An advantage of the D–ISP that no cache can provide is that it needs no associative cache hit detection during function execution. Instead of this only a simple address translation has to be performed (an asynchronous fetch process implementation requires 16 ALUTs and 2 registers as shown later in Section 6.1.4). The lightweight operation during function execution may be beneficial in terms of power consumption, because it is known that caches are a major power consumer within the processor (depending on size, organisation, and process the numbers range from 15% to 50% of the overall power dissipation for the cache subsystem [Montanaro et al., 1996; Malik et al., 2000; Benini et al., 2001; Clark et al., 2001; Moshnyaga, 2001; Qadri et al., 2009]). In embedded systems scratchpads are used to reduce the dissipated energy of the system (e.g. by [Kandemir et al., 2001; Banakar et al., 2002; Vander Aa et al., 2003; Janapsatya et al., 2006a]), because the expensive lookup logic of an associative cache is not needed (refer to [Reinman and Jouppi, 2000] for the impact of associativity on the cache energy consumption in CACTI and to [Zhang et al., 2005] for its implication on cache design). Hence, it is likely that the D–ISP has a lower power dissipation than a conventional cache. So a deeper look in the energy efficiency of the D–ISP and a comparison to caches and software-managed scratchpads is worth for future work.

As shown the D–ISP requires a significantly higher amount of hardware resources than a common instruction cache. An implementation of the real-time capable CarCore processor uses in the single threaded version 14,808 ALUTs and 3,857 registers, according to [Mische et al., 2010a]. Embedding the D–ISP into the CarCore processor the D–ISP with a lookup width of 4 functions requires about 5.2% and 13.7% of the resources of the CarCore with D–ISP for ALUTs and registers, respectively. This is a rather large fraction of the processor, but the instruction memory hierarchy has a high impact on the overall system performance. Furthermore, the design goals of D–ISP are the providing of low WCET estimates and also the ease of the low-level system analysis. To reach these aims compromises in terms of increased hardware effort had to be made that cause the D–ISP to be more complex than a traditional cache. But, as it will be shown in Section 6.2, the additional hardware effort of the D–ISP pays off in terms of lower WCET estimates than common instruction memories can reach.

6.1.3 Memory Overhead

To quantify the memory overhead of the D–ISP that is originated by the helper memories in comparison to the tag memory of a cache the overall memory sizes of both types is examined. For the calculation of the sizes of the D–ISP helper memories the Equations (3.4) (*mapping table memory*, MTM), (3.8) (*lookup table memory*, LTM), and (3.9) (*context stack memory*, CSM) from Section 3.3.4 were used. The overall memory usage of the D–ISP (mem($D\text{–}ISP$)) is the sum of the size of the D–ISP and all its helper memories:

$$
\begin{aligned}
\text{size}(MTM) &= \left(width_{addr_N} + width_{addr_D} + \left\lceil \log_2(\max(\text{size}(function))) \right\rceil\right) \cdot no_{func} \\
\text{size}(LTM) &= width_{addr_N} \cdot no_{func} \\
\text{size}(CSM) &= width_{addr_N} \cdot no_{stack\_depth} \\
\text{mem}(D\text{–}ISP) &= \text{size}(D\text{–}ISP) + \text{size}(MTM) + \text{size}(LTM) + \text{size}(CSM)
\end{aligned}
$$

Except of the number of functions that can be maintained by the D–ISP concurrently (no_{func}) and the scratchpad size (size($D\text{–}ISP$)) all parameters are taken as fixed. The chosen values for the D–ISP are shown in Table 6.5. The width of the addresses in normal address space ($width_{addr_N}$) that have to be stored in all D–ISP helper memories is set to 28 bit, because the TriCore address space is divided into 16 segments [TriCore, 2007a]. Thus, the highest 4 bit of the address is chopped off, because it is assumed that the whole application code is located

Table 6.5: Architectural parameters of the D–ISP helper memories

Parameter	Description	Value
$width_{addr_N}$	Address width in normal memory	28 bit
$width_{addr_D}$	Address width in scratchpad memory	16 bit
max(size($function$))	Maximum supported function length	64 KiB (2^{16} B)
no_{stack_depth}	Maximum supported stack depth	16
no_{func}	Number of maintainable functions	8 - 256
size(D–ISP)	Size of the scratchpad memory	128 B - 32 KiB

within one code segment. The scratchpad's address space is 16 bit wide ($width_{addr_D}$), allowing a maximal scratchpad size of 64 KiB, which is assumed to be suitable for an instruction memory for embedded systems. This upper-bounds the maximum supported function size also to 64 KiB, which implies that 2 B can be used to store the function's size in the *mapping table memory* (see Equation (3.4)). A maximum stack depth of 16 is assumed, meaning that the *context stack memory* has 16 entries. If an application requires a deeper context stack, either a larger *context stack memory* is needed or the D–ISP can be disabled before a stack overflow occurs. The latter case is described in Section 5.3.1 with more detail.

To compare the memory overhead of the D–ISP with a common cache memory the size of the tag memory of a cache has to be quantified. Therefore, it is assumed that the maintenance structures of the cache are located in memory and not embedded as logical registers in the controller. According to [Patterson and Hennessy, 2005, Chapter 7.3] and [Hennessy and Patterson, 2006, Appendix C] the cache address is partitioned into a tag, index, and a block-offset. The block-offset is used for intra-cache-line addressing, the index is to select the correct set, and the tag part of the address is needed for hit/miss detection. Hence, only the tag part of the address needs to be hold by the cache. So the memory overhead of a cache is delimited by the number of cache lines and the width of the tag part of the address ($width_{addr_{tag}}$). In addition to that for every cache line a valid bit has to be maintained. The equations[11] to calculate the size of a cache tag memory (size(TAG)) and the overall memory usage of a cache (mem(I–$Cache$)) are shown below:

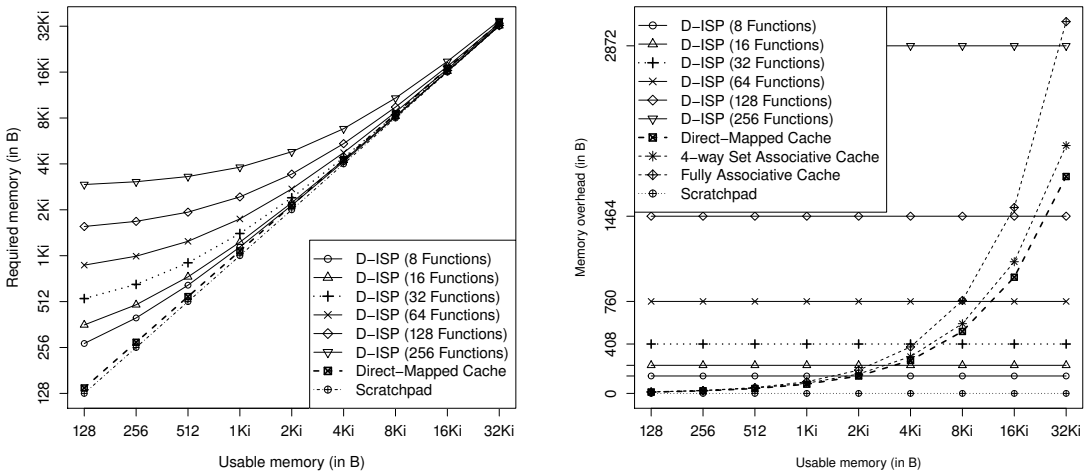
$$
\begin{aligned}
width_{addr_{blockoffset}} &= \log_2(\text{size}(CL)) \\
width_{addr_{tag\&index}} &= width_{addr_N} - width_{addr_{blockoffset}} \\
width_{addr_{index}} &= \log_2(\text{size}(I\text{–}Cache)) - width_{addr_{blockoffset}} - \log_2(associativity) \\
width_{addr_{tag}} &= width_{addr_{tag\&index}} - width_{addr_{index}} \\
\text{size}(TAG) &= \frac{\text{size}(I\text{–}Cache)}{\text{size}(CL)} \cdot (width_{addr_{tag}} + 1) \\
\text{mem}(I\text{–}Cache) &= \text{size}(I\text{–}Cache) + \text{size}(TAG)
\end{aligned}
$$

For the comparison to the memory overhead of the D–ISP the same address width of 28 bit is assumed for the cache. The other parameters needed by the equations are shown the Table 6.6. The cache line size is assumed to be 32 B and thus as large as the line size of the caches in the TriCore processor [TriCore, 2004]. As size for the cache values up to 32 KiB are assumed. For the comparison to the memory overhead of the D–ISP three cache organisations are used: direct mapped (associativity of 1), 4-way set associative, and fully associative (associativity of the number of cache lines, #CL).

[11] The 3rd equation equals to $2^{\text{index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativitiy}}$ from [Hennessy and Patterson, 2006, App. C].

Table 6.6: Architectural parameters of the I–Cache

Parameter	Description	Value
$width_{addr_N}$	Address width in normal memory	28 bit
size(CL)	Cache line size	32 B
size($I\text{–}Cache$)	Size of the cache memory	128 B - 32 KiB
$associativity$	Associativity of the cache	1, 4, or #CL



(a) Used memory including helper/tag memory

(b) Overhead of the helper/tag memory

Figure 6.2: Memory effort of D–ISP in comparison to direct mapped cache and scratchpad for different memory sizes

The Figure 6.2 shows the overall memory usage and the memory overhead of the D–ISP with different table sizes (no_{func}), the three cache configurations, and a scratchpad memory. It can be seen that an ordinary scratchpad memory has no memory overhead, since it is managed by software and thus does not need any additional maintenance structures. Figure 6.2(a) shows that a cache memory[12] needs only a small additional memory space for maintaining the tag memory and the valid bits. In contrast to this the overall memory usage of the D–ISP including the size of the helper memories is huge compared to the scratchpad memory size, especially for very small memory sizes. This is because the table sizes in the helper memories are independent of the size of the scratchpad memory. Hence, for increasing memory sizes the overhead decreases relative to the memory size. The constant helper memory size of the D–ISP is exposed in Figure 6.2(b): For each number of supported functions (no_{func}) a constant memory overhead is shown. In contrast to this for the cache memories the size of the tag memory increases, because when larger cache sizes are used and the cache line size (size(CL)) is fixed, the number of cache lines has to raise. So the tag memory size increases linear to the cache memory size.

Selecting the number of supported functions (no_{func}) for the D–ISP depends not only on the amount of memory overhead, also the expected function sizes have to be taken into account.

[12]In the Figure 6.2(a) the small differences of the tag memory sizes for different associativities are not visible. Therefore, only a direct mapped cache is shown.

For example a D–ISP with a size of 128 B can never hold 128 functions concurrently, because even the return instruction of the CarCore (TriCore ISA) is 2 bytes long. Hence, a large number of concurrently supported functions for small D–ISP sizes does not make sense. On the other hand having too small function tables for a large D–ISP size may result in the frequent eviction of functions caused by mapping/lookup table overrun. Therefore, a good choice of the ratio of management table size and D–ISP memory size is of importance. Holzmann [2006] suggests that the size of a function should be limited to one page of code, because this eases the comprehensibility and verification of the functions and so the whole application code. Hence, for a D–ISP with a table size of 32 functions a memory size of 4 KiB to 32 KiB is suitable, because then an average function size between 128 B and 1,024 B can be reached, if 32 functions are contained in the D–ISP memory. Comparing the memory overhead of the cache configurations in Figure 6.2(b) with the 32 function D–ISP it is shown that above a cache size of 8 KiB the overhead of the caches is actually higher than the overhead of the D–ISP. So when the D–ISP memory size is properly dimensioned concerning to the expected sizes of the functions, a smaller memory overhead than a cache requires is reachable.

Usually when increasing the size of a cache also the cache line size is increased, to reduce the size of the tag memory. But the proper balance of cache size and cache line size is dependent on the characteristics of the system and also on the application domain. Hennessy and Patterson discuss the effect of increasing the cache line size and concluding in [Hennessy and Patterson, 2006, Appendix C.3] that larger cache lines will reduce the miss rate due to the better exploitation of the spatial locality. But on the other hand the miss penalty can be increased, because larger memory blocks have to be loaded. How much the miss penalty is increased by larger cache lines is dependent on the memory system. Furthermore, if the cache line size is too large, even the miss rate might be increased, due to a higher chance of conflict misses. As rule of the thumb Hennessy and Patterson suggest that for systems with low latency and low bandwidth small cache lines are preferred, since the benefit of larger blocks is low in such systems. Anyhow, the optimal cache line size is not easy to determine. Many work is done to find the optimal cache line size for different memory systems and applications as e.g. in [Smith, 1987; Sheu et al., 1990; Shiue and Chakrabarti, 1999; Srinivasan et al., 2001; Milenkovic et al., 2003; D'Alberto et al., 2005]. Therefore, the cache dimensions of the TriCore processor, which is 32 B for the cache line and cache sizes between 4 KiB and 16 KiB [TriCore, 2004], are assumed for the quantification of the memory overhead (and all other evaluations).

It is shown that the memory overhead caused by the D–ISP helper memories depends on the number of maintainable functions (and also the size of the *context stack*) and is not affected by the size of the scratchpad memory. Furthermore, for a proper ratio of the D–ISP size and the number of supported functions the D–ISP has a comparable memory overhead as caches of different associativity, as Figure 6.2 shows.

6.1.4 Timing Characteristics

When looking at the timing characteristics of the D–ISP two aspects are highlighted in the following: the maximal clock rate of the D–ISP controller and the timing implications of the D–ISP on the fetch path.

Maximum Clock Rate of the D–ISP Controller

The Table 6.7 shows the maximum frequency that the D–ISP controller is able to run depending on the number of functions that can be looked up within one cycle (no_{look}). It can be seen that until a lookup width of 32 functions the maximum frequency is mainly at the level of about

Table 6.7: Maximum frequency of the D–ISP controller for the Stratix II EP2S180F1020C3

Lookup Width no_{look} (in Functions)	**Maximum Frequency of the D–ISP Controller**
2	95.20 MHz
4	**103.17 MHz**
8	100.44 MHz
16	101.41 MHz
32	102.60 MHz
64	85.34 MHz
128	72.90 MHz
256	50.97 MHz

100 MHz. Above 32 the maximum frequency of D–ISP controller drops and reaches 51 MHz at a lookup width of 256 functions. This frequency drop is explainable when looking at the logic amount of the D–ISP controller. The Figure 6.3 sets the development of the number of ALUTs of the D–ISP controller in relation to its maximum frequency. It shows that the frequency drop goes along with the increase of used ALUTs for D–ISP controller implementations with larger lookup widths. This is due to the fully associative lookup table of the D–ISP that requires a gain of the logic amount for increasing look up widths. In the D–ISP controller the lookup of no_{look} functions is done within one cycle. Thus for a larger lookup width more ALUTs have to be placed on the timing critical path of the implementation and also the signal routing in the FPGA gets more complicated. Hence, the maximum frequency of the D–ISP controller declines with an increase of the function lookup width.

The Figure 6.3 also shows a low maximum frequency for 2 functions, a outlying high for 4 functions, and a slightly upward slope of the frequency for a lookup width between 8 and 32 functions. It is assumed that this variations are caused by the optimisations of the signal routing during synthesis of the D–ISP controller.

Timing of the Fetch Control and Impact on the Fetch Path

In Section 3.3.3 two possibilities of implementing the D–ISP *fetch control* is discussed. At first a synchronous implementation of the *fetch control*, in which the translation of the native fetch address into the D–ISP address space takes one cycle, is considered. This adds one additional cycle fetch latency to accesses to the D–ISP. By implementing the *fetch control* in a pipelined manner, in which always one fetch request is buffered, one fetch request can be handled per cycle. So an fetch throughput of 1 is reached.

The other possibility of implementing the *fetch control* is to calculate the D–ISP address asynchronously. In contrast to the synchronous implementation every D–ISP fetch access is handled in one cycle, but an additional delay is introduced on the fetch path. This means that more logic is planted on the path from checking if a fetch request is needed up to the delivering of the address to the scratchpad memory. That will result in a higher path delay. Then if this path turns out to be the critical path of the host processor (if it is not already the critical path), the use of the D–ISP has an impact on the maximum possible clock frequency of the whole processor.

To check this influence the *fetch control* process was separated from the rest of the D–ISP controller. The synchronous implementation of the *fetch control*[13] reaches a maximum clock frequency of 178.86 MHz, requires 150 ALUTs, and 84 registers on the Stratix II EP2S180F1020C3

[13]The D–ISP controller implementation that was examined in Section 6.1.2 uses the synchronous *fetch control*.

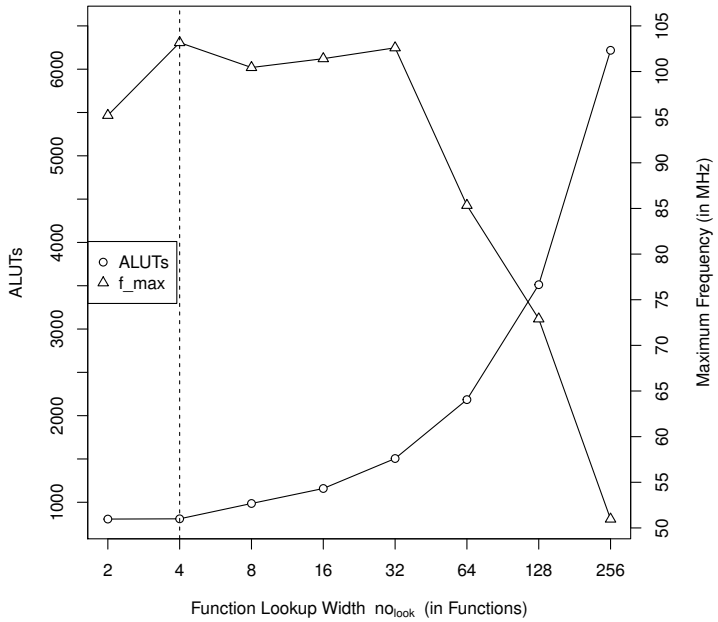


Figure 6.3: Resource usage and maximum frequency of the D–ISP controller depending on the lookup width for the Stratix II EP2S180F1020C3

FPGA. The rather large amount of logic needed by the simple *fetch control* is caused by the management of the buffered fetch requests that is needed to allow a pipelined D–ISP access. Also the critical path in the synchronous *fetch control* process is dominated by the buffer management.

The asynchronous implementation of the *fetch control* by splitting the process into an asynchronous address calculation and an synchronous address check logic (according to Figure 3.10 in Section 3.3.3) delivers a logic usage of 16 ALUTs and 2 registers. The low amount of logic needed to implement both subprocesses results in a negligible delay time for the asynchronous address calculation. So the impact on the fetch timing path induced by the asynchronous D–ISP *fetch control* can be considered as insignificant.

Nevertheless, if every change of the timing characteristics of the host processor is unwanted or prohibited e.g. by safety or verification constraints, the synchronous *fetch control* is to be used. It requires additional cost in terms of logic and registers, especially for the pipelining of the fetch requests that allows the D–ISP with a synchronous implementation of the *fetch control* to reach the same throughput of fetches as the asynchronous *fetch control*. Thus the fetch access time is delimited by the access time of the scratchpad memory only. Hence, the D–ISP can serve fetches as fast as common on-chip scratchpad memories that do not feature a dynamic memory management implemented in hardware.

6.1.5 Hardware Effort of the D–ISP with Stack-Based Replacement

Until now all discussions in this section considered the D–ISP with a FIFO replacement policy, but in Section 3.2.4 also a stack-based replacement policy is introduced and further in Section 3.3.4 a

Table 6.8: Comparison of the resource usage of FIFO and stack-based D–ISP for Stratix II EP2S180F1020C3

Replacement Policy (Lookup Width no_{look})	**ALUTs**	**Registers**	**Maximum Frequency of the D–ISP Controller**
FIFO (4)	809	613	103.17 MHz
Stack-Based (4)	995	643	101.21 MHz
Difference of Stack-Based Policy Compared to FIFO (in Percent)	**+23.0%**	**+4.9%**	**-1.9%**
FIFO (32)	1,505	665	102.60 MHz
Stack-Based (32)	1,648	698	96.66 MHz
Difference of Stack-Based Policy Compared to FIFO (in Percent)	**+9.5%**	**+5.0%**	**-5.8%**

possibility of its implementation is discussed. Due to the need of two write and eviction pointers, the differences in accessing the D–ISP helper memories, and slightly changed operation (e.g. the loading of functions in reverse order on return) the needed amount of hardware is larger than for the FIFO implementation. The Table 6.8 gives an overview of the hardware complexity and the maximum frequency of the D–ISP controllers with a lookup width of 4 and 32 functions for FIFO and the stack-based replacement policy. Notice that the sizes and organisation of the helper memory is not affected by the change of the replacement policy.

It can be seen that the additional hardware amount is about 10-23% and 5% for ALUTs and registers, respectively. Additional logic elements are needed by the different operation modes for call and return, if the function has to be loaded into the D–ISP. Also the maintenance of the different write and eviction pointers requires additional logical elements. The relative reduction of the ALUT overhead for a larger lookup widths shows that the additional logic needed for the stack-based policy is independent of the lookup width. The increased amount of logic registers is mainly caused by the two extra pointers and their maintenance. The number of the additionally required registers is also nearly constant and ranges about 30 registers for different lookup widths.

To keep the additional hardware amount low for the implementation of handling misses on return the controller needs to load the first block of the function twice: first to determine the size and the end address of the function to load and second to store the block into the scratchpad memory. Thus the load of a function on miss needs one extra off-chip memory access. This additional latency could be saved, if the address of the first block in the scratchpad is calculated directly after decoding the function length and the block is written into the scratchpad. In fact on writing the block it has to be checked that no other function is overwritten. This is not trivial, because of the structure of the eviction pointers, only the function that is evicted next is known by the D–ISP controller (refer to Figure 3.15 in Section 3.3.4) and the function that might be evicted by writing the first block of the function might not be the next leftmost function stored in the scratchpad. Thus the implementation of a direct write of the first function block on return miss is not suitable in terms of additional required logic. Another way to get rid of the additional off-chip memory access is to buffer the 64 bit fetch word on its first load and then write it into the scratchpad, if the write pointer reaches the corresponding position. To keep the hardware cost for the stack-based policy as low as possible the buffering of the first fetch block is not implemented.

Another difference to the FIFO replacement policy is that the stack-based replacement policy requires that the off-chip memory access time is at least 4 processor cycles instead of 2. This is because in the worst case when evicting functions of the size of one block on return the

D–ISP controller needs to write the *mapping table* to invalidate and determine the address on which the next function is overwritten. This return eviction address is calculated by the address of the function in the scratchpad and its size, since on return the last block of a function is overwritten first. As the connection of the D–ISP controller to the *mapping table* consists of one read and one read/write port with both a width of half a *mapping table* entry, the eviction of one function and the determination of the start address of another within one cycle is possible. But because the stack-based policy needs also the size of the function to determine the return eviction address, an additional access to the *mapping table* is needed. So until these two table accesses are completed no new incoming fetch block can be handled by the D–ISP controller resulting in a minimal off-chip memory access time of 4 cycles[14]. A wider read port for the *mapping table* or an additional one would reduce the minimal off-chip memory latency back to 2 cycles. Anyhow, the latencies of the off-chip memory (or any other next memory level) are usually higher than 2 or 3 processor cycles: For a general purpose system the memory latency of on-chip caches range about hundreds of picoseconds, whereas the latency of off-chip caches is in the magnitude of nanoseconds [Jacob et al., 2007]. Also for embedded systems the memory latency of the off-chip memory cannot be ignored [Milenkovic et al., 2003; Marwedel, 2007]. Therefore, the support of a minimal off-chip memory access time of 4 cycles is suitable for the D–ISP controller with stack-based replacement policy. Such that the bandwidth of the connection to the helper memories does not need to be increased.

The WCET evaluation of the stack-based replacement policy in Section 6.2.6 will show that the additional hardware effort needed by the stack-based replacement policy is worth, especially for small scratchpad memory sizes.

6.1.6 Conclusion

The D–ISP controller requires a higher logic amount than a common cache. This is mainly due to the more complex internal processing of the controller that requires not only to compare tags and detect a miss. The D–ISP controller has also to initiate the load of a complete function and therefore determine the function's size. So the hardware complexity of the baseline version of the D–ISP controller (with a lookup width of 4) is in the same magnitude as an LRU cache with a higher associativity, as the comparison to Table 6.3 shows. With higher lookup widths the complexity of the D–ISP controller increases significantly, because more comparisons have to be performed within one cycle. Moreover, the increased hardware effort for high lookup widths might be gained by the expensive representation of associative structures in an FPGA as figured out by Wong et al. [2011].

The D–ISP's memory overhead strictly depends on the number of functions that can be kept concurrently in the scratchpad. In a reasonable range of the scratchpad size for the number of supported functions, e.g. 32 functions and a D–ISP size of above 4 KiB, the memory overhead of the D–ISP is comparable or even lower than the size of the tag memory a cache needs. By the separation of the scratchpad memory and the lookup tables the D–ISP decouples the content and the maintenance memory structures. This is advantageous, because it allows a fine-grained scratchpad memory with a low amount of internal fragmentation without an increased memory overhead. In contrast to the D–ISP the size and number of cache lines (and also other parameters) define the size of tag memory and so the memory overhead of the cache.

The maximum clock frequency of the D–ISP controller is basically about 100 MHz with a lookup width of up to 32 functions. In [Mische, 2011] is shown that the maximum frequency of the CarCore is below 30 MHz. Hence, the D–ISP does not affect the maximum clock frequency

[14]Notice the access time of the helper memories is 1, such that data arrives at the D–ISP controller two cycles after requesting it.

of its host processor. By the lightweight fetch control process of the D–ISP controller only a negligible additional delay is added to the fetch path of the processor. Therefore, fetch requests to the D–ISP can be served within the same time as memory accesses to common scratchpads. In contrast to this the cache's hit detection adds additional delay onto the fetch path, causing higher hit latencies for caches with higher associativity (refer e.g. to [Hill, 1988; Hennessy and Patterson, 2006, Section 5.2]).

To summarise, the D–ISP poses a higher hardware cost than a common cache memory, but it also has beneficial characteristics that cannot be given by any cache, for example the decoupling of content and management memory structures or its minimal impact on the timing of the fetch path.

6.2 Impact on the Worst Case Execution Time

In this sections various facets of the worst case behaviour of the D–ISP are discussed. At first in Section 6.2.1 the evaluation methodology is introduced including the timing model of the processor, the different instruction memories to which the D–ISP is compared to, and the chosen benchmark set. Section 6.2.2 gives a short discussion of the different S–ISP assignment algorithms. Based on these results the best performing assignment algorithm is used as opponent for the D–ISP. The examination of the WCET estimates for the D–ISP is provided in Section 6.2.3. Furthermore, in this section the D–ISP is compared to other instruction memories. The overhead which is caused by the dynamic content management of the D–ISP is highlighted separately in Section 6.2.4. Section 6.2.5 discusses the impact of the interference penalty on the WCET estimate of the instruction cache. The influence on the estimated WCET caused by the different D–ISP replacement policies, which were proposed in Section 3.2.4, is investigated in Section 6.2.6. Finally, Section 6.2.7 shows the design point decision of the D–ISP implementation in which the hardware complexity is traded against the estimated WCET performance with the function lookup width as design parameter. Parts of the following section were published in Metzlaff and Ungerer [2012a; 2012b].

6.2.1 Evaluation Methodology

To quantify the WCET impact of the D–ISP the static timing analysis tool ISPTAP, presented in Section 5.2, was used. The analysed code is instrumented with the FLE instructions by the tool presented in Section 5.1.

Architectural Configuration

As processor model for the WCET analysis the CarCore with the pipeline timing model presented in Table 5.1 (p. 142) is used. The Maximum Memory Access Times (MMATs) that are used in the evaluation are shown in Table 6.9. Two configurations are distinguished: one in which the off-chip memory connection is separated for instruction and data access (denoted further as SEMC) and one that employs a shared connection. If a shared off-chip memory connection (also denoted as SHMC) is considered, the memory latencies for off-chip accesses have to be increased, because of possible collisions of data and instruction memory accesses. To be safe it is assumed that every off-chip memory access may suffer this interference. For a less pessimistic assumption the memory and pipeline analysis has to be integrated at the cost of a higher analysis complexity. A more precise discussion of the impact of the memory connection on the WCET analysis is given by Section 2.7.

Table 6.9: Maximum Memory Access Times (MMATs) for the different memory types

Static Memory	**Separated Connection**		**Shared Connection**	
On-Chip Scratchpad	1		1	
Off-Chip Data/Instruction	4		7/8	
Dynamic Memory	**Hit**	**Miss**	**Hit**	**Miss**
I–Cache (32 B lines)	1	17	1	33
D–ISP (8 B blocks)	1	4	–	–

For all on-chip memory accesses, no matter if they are directed to static scratchpads or hit the instruction cache or the D–ISP, the MMAT is one cycle. An access time of one cycle means that the requested data is delivered one cycle after requesting it, i.e. a latency of zero clock cycles.

In Table 6.9 two latencies are given for the off-chip memory with a shared off-chip memory connection: The lower value is for data memory accesses, because in the case of that an instruction and a data memory access is scheduled in the same cycle, the memory arbiter will select the data memory access. Thus the data access wins the arbitration on conflict with a instruction memory access. (The favouritism of the data memory accesses is motivated by the general observation that, if a data memory access is delayed, the pipeline will be stalled[15] and the prioritised fetched instructions cannot be issued to the pipeline.) So, on concurrent request of data and instruction memory the data memory access is always preferred. Non-concurrently issued memory requests are processed by the arbiter in a FIFO manner. Therefore, a data memory access can be delayed by at most $(MAT - 1)$ cycles[16], because an instruction memory access could be scheduled in the cycle before the data memory access is ready. Since the instruction memory access can be delayed by a data memory access, if both are requested in the same cycle, its MMAT is higher. Notice that only an interference of one instruction memory access and one data memory access is possible, since the requests are handled by the memory controller in order and another memory request can only be released, if the previous of the same type is finished. So at most one instruction and one data memory access can be pending at any point in time[17].

The memory access times for cache misses in Table 6.9 were calculated as follows: the CarCore processor supports at most 64 bit memory accesses. Therefore, the 32 B cache line is to be loaded by 4 subsequent memory accesses, resulting in 4 times the off-chip memory access time. The cache line size of 32 B is chosen according to the instruction cache specifications of the TriCore processor [TriCore, 2004]. In addition to the memory access time for loading the instructions into the cache one additional cycle is needed by the cache for the miss handling. As the access time for the separated off-chip memory connection is 4 cycles, a cache miss is handled within 17 cycles. According to the memory access time for instruction memory accesses when a shared off-chip memory connection is used the cache miss takes is 33 cycles[18]. Notice that the cache lines are not loaded using one burst load that would be faster than four 64 bit accesses, because this would increase the worst-case memory access time for the (not cached) data memory accesses, since

[15] The pipeline of the CarCore cannot stall, but if a memory access from a thread is delayed it has to wait until the access is completed and the insert load instruction can be issued. Refer to Section 5.2.2 or [Mische et al., 2010a] for the load handling in the CarCore.

[16] MAT – Memory Access Time

[17] For the SMT capable CarCore this restriction holds per thread.

[18] Every 64 bit load can interfere with a data memory access in the worst case.

Table 6.10: D–ISP helper memory sizes and activation time parameters used in WCET analysis

Helper Memory	**Size in Functions**	**Refer to Constant**
Function Table	32	no_{func}
Lookup Table	32	no_{look}
Max. Stack Depth	10	no_{stack_depth}
Activation Time	**in Cycles**	**Refer to Equation**
Hit	4	(3.10)
Miss	$5 + 4 \cdot \text{size}(function)$	(3.11)

Table 6.11: Jump and size penalties for the BBS–ISP memory that are applied by ISPTAP

Jump Penalty Term	**Penalty in Cycles**
jp_{ca}	$3 + 3 \cdot MMAT$ cycles
jp_{js}	$1 \cdot MMAT$ cycles
jp_{jl}	0 cycles
jp_{cs}	$1 \cdot MMAT$ cycles
jp_{cl}	0 cycles
Size Penalty Term	**Penalty in Bytes**
sp_{ca}	4 B
sp_{js}	2 B
sp_{jl}	0 B
sp_{cs}	2 B
sp_{cl}	0 B

they can be delayed by a burst load on instruction cache miss. Thus for better comparability with the scratchpad memories the loading of cache lines is performed by four independent loads.

One major contribution of the D–ISP is that the memory access time on loading a function is independent of the off-chip memory connection type that is applied. This is due to the two-phased execution scheme that is enforced by the usage of the D–ISP (refer to Section 3.1). Therefore, the same off-chip instruction memory access time can be used for a configuration with separated and with shared off-chip memory connection. This holds also for the latency needed by any data memory access. Because the D–ISP loads only complete functions into the scratchpad, the miss latency is dependent on the function size. To load one 64 bit block 4 cycles are needed. As described in Section 3.3 the D–ISP needs additional time for hit/miss detection on function activation. Table 6.10 shows the relevant parameters of the D–ISP assumed for the WCET analysis.

In Section 4.1.3 the jump penalties for the BBS–ISP memories when reconnecting the basic blocks after assigning them to the scratchpad were introduced. The Table 6.11 provides the used penalties for the WCET analysis. The table contains the jump penalties according to Equation (4.17) and the size penalties as introduced by Equation (4.18). In addition to the proposed size penalty in Section 4.1.3 the TriCore instruction set [TriCore, 2007b] supports short calls with an instruction length of 16 bit. Therefore, the additional class *cs* is shown in Table 6.11.

The jump penalty when two basic blocks were connected by continuous addressing (jp_{ca}), which requires to insert a jump, is calculated ass follows. According to Table 5.1 a jump is accounted with 3 cycles. In addition to that the IW will be flushed by the CarCore, if a basic block is entered by jumping. Because the IW has a size of 3 (as described in Section 5.2.2),

Table 6.12: Benchmarks used for WCET analysis with ISPTAP (* including a benchmark wrapper function and depending on the benchmark a harness and/or an initialisation function, [1] the main function is encapsulated in a loop that runs 8 times, [2] to remove code redundancy the code of the main function is encapsulated in a function)

Benchmark	**Code Size*** **in Byte (B)**	**Largest Function in Byte (B)**	**Fraction of Largest Function in Code**	**Function Count***
Adpcm†	2,744	742	27.0%	17
Compress†	1,292	348	26.9%	9
Dijkstra§	816	396	48.5%	9
Edn†	1,262	610	48.3%	9
Edn *Loop*[1]	1,304	610	46.8%	10
Matmult†	310	94	30.3%	6
Matmult *Loop*[1]	352	94	26.7%	7
Puwmod‡	1,272	1,176	92.5%	3
Puwmod *Split*[2]	948	784	82.7%	4
Rspeed‡	710	614	86.5%	3
Rspeed *Split*[2]	474	310	65.4%	4
Sha§	1,272	366	28.8%	14
Ttsprk‡	4,150	3,382	81.5%	5
Ttsprk *Loop*[1]	4,196	3,382	80.6%	6
Ttsprk *Split*[2]	2,542	1,706	67.1%	6

three times the Maximum Memory Access Time (MMAT) to fill the IW has to be accounted. In the timing model used by the ISPTAP any jump takes the same amount of time and the effect of additional instructions caused by preparing a far jump (e.g. by writing a part of the jump address in a register) is not considered. The TriCore instruction set supports 16 and 32 bit instructions, at which the 16 bit branch instructions can only target nearby addresses[19]. Because the static scratchpad will be located in another memory section the 16 bit instructions have to be extended to 32 bit, resulting in a size penalty of 2 bytes for every 16 bit branch instruction that connects the off-chip memory address space with the scratchpad and vice versa. By increasing the size of short jumps and short calls it is possible that another fetch word has to be requested to execute the altered basic block. Therefore, one additional fetch request of $1 \cdot MMAT$ cycles is accounted as penalty of short jumps and short calls.

Benchmarks

The benchmarks used for the WCET evaluation are characterised in Table 6.12. The selected benchmarks are from the following three different benchmark suites: Mälardalen WCET benchmark suite [Gustafsson et al., 2010] (marked with †), EEMBC AutoBench 1.1 [Poovey et al., 2009] (marked with ‡), and the MiBench suite [Guthaus et al., 2001] (marked with §). Beside the code size also the size of the largest function and its contribution to the benchmark is shown. Both are of importance for the D–ISP, because the size of the largest function denotes the minimum possible scratchpad size. In addition to that the number of functions in the benchmark is shown in Table 6.12. The number of functions in the benchmark includes the harness and/or initialisation functions the benchmark has, except any functions that are not reachable from the benchmark's main function. As the number of functions is below 32 for all benchmarks no

[19] Depending on the particular instruction the displacement is between 9 and 5 bit [TriCore, 2007b].

mapping table overrun may occur. Thus the effect of function eviction due to the limited size of the D–ISP helper memories is not shown in this evaluation.

For several benchmarks a second configuration was analysed, that is denoted by *Loop*. In this configurations the benchmark's entry function is placed in a loop with 8 iterations, to check if any trashing effects occur for the D–ISP. Another additional configuration, denoted with *Split*, is created for the EEMBC benchmarks, to reduce the size of the largest function. The EEMBC benchmarks usually consist of a main function in which the same code is executed three times, i.e. a loop of three iterations is unrolled in the benchmark's code. This results in a large main function with redundant code that is awkward for the D–ISP, because the minimal size of the D–ISP has to be unnecessarily large and an eviction of such a large function is extremely costly. The code is also not optimal for caches, since the redundant code renders a bad cache footprint, such that the cache cannot exploit the application's locality and tends to suffer thrashing. Therefore, the code of one iteration is encapsulated in an additional function, which is called three times with the correct context by the main function of the benchmark, i.e. the unrolling of the loop that was made by the original code is revoked. Thus it is possible to reduce the overall code size, the size of the largest function, and its contribution to the application's code size. Depending on the complexity to distinguish the different contexts the size of the function that encapsulates the iteration varies. Notice that due to the additional function the number of instructions that have to be executed is increased: first the function has to be called and second the additional overhead due to distinguishing the execution context of the function has to be considered. However, the *Split* configuration is build for the EEMBC benchmarks, because it allows to reduce the minimal D–ISP size to execute the benchmark and also promises a better utilisation of the on-chip instruction memories.

Beside the changes in the *Loop* and *Split* configurations the used benchmarks were not altered beyond the adjustments that are needed to execute the code with the CarCore processor. Because the CarCore is multi-threaded, the usage of global variables were forbidden to ensure a thread-safe execution. Therefore, all global variables are transferred into a structure that is thread-local. Furthermore, the main benchmark function was embedded in an thread harness to separate benchmark independent initialisation code. Minor changes were also necessary for the EEMBC benchmarks in which the benchmark function also contains the parameter initialisation. So to analyse only the real benchmark code it was required to split the initialisation and the benchmark code into different functions. For all benchmarks the function that starts the benchmark code itself is used as entry point to the WCET analysis. For the *Loop* benchmark configurations the entry function for the analysis is a function that calls the benchmark function in a loop.

Another exception is the Sha benchmark that was altered to stress the D–ISP with a higher number of functions and a more complex call graph. Therefore, six macro definitions (`f1`, `f2`, `f3`, `f4`, `ROT32`, and `FUNC`) were converted into small functions. The execution of the functions `f1` to `f4` is data dependent and which of them will be called is decided within a loop (see also Algorithm 6.1 in the next section). So due to these changes the number of function in the Sha benchmark is extended from 8 to 14.

All analysed benchmarks were compiled by the HighTec TriCore GCC 3.3 [HighTec, 2003] with the compile flags as shown in Section 5.3.2. Notice that the compiler is allowed to use the optimisation level 2, which enables nearly all supported optimisations of the TriCore GCC. A matrix of all analysed benchmarks and the types of memory that were examined is shown in Table 6.13. Nearly all shown configurations are analysed by ISPTAP within a reasonable time. For a closer look on the analysis time and complexity refer to Section 6.4.

Table 6.13: Benchmark instruction memory matrix: ✓ – configurations ran successfully in all memory sizes, S – for at least one memory size the ILP solver needs more than 1 h to find the assignment, E – state space explosion of DFA leading to summarized analysis time above 1 h for all benchmark configurations (KN: knapsack-based, KNJ: knapsack-based with jump and size penalties, WS: WCP-sensitive, WSJ: WCP-sensitive with jump and size penalties, DM: direct mapped, FF: FIFO replacement policy, LR: LRU replacement policy, ST: stack-based replacement policy)

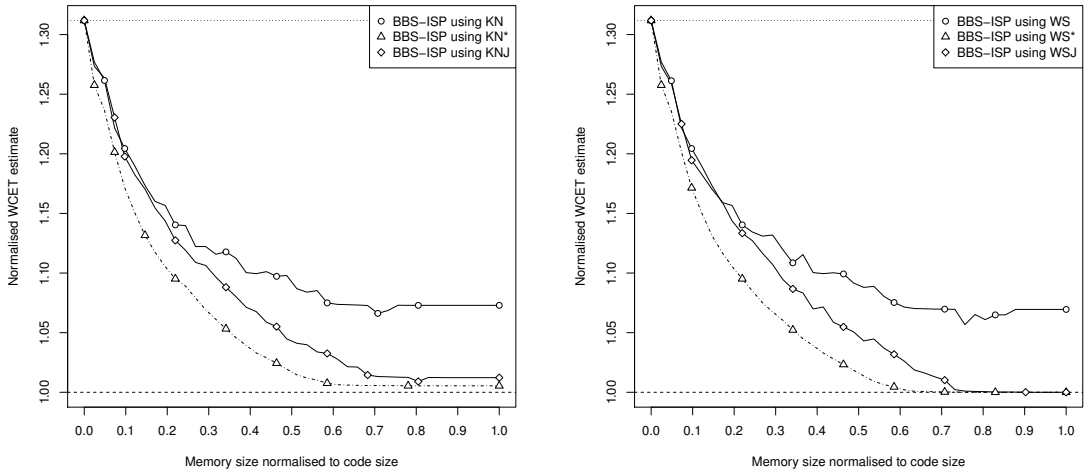
Benchmark	**BBS–ISP**				**FS–ISP**		**I–Cache**			**D–ISP**		
	KN	KNJ	WS	WSJ	KN	WS	DM	FF	LR	FF	LR	ST
Adpcm†	✓	S	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Compress†	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Dijkstra§	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Edn†	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Edn *Loop*[1]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Matmult†	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Matmult *Loop*[1]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Puwmod‡	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Puwmod *Split*[2]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Rspeed‡	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Rspeed *Split*[2]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Sha§	✓	✓	✓	✓	✓	✓	✓	E	✓	✓	✓	✓
Ttsprk‡	✓	S	✓	S	✓	✓	✓	✓	✓	✓	✓	✓
Ttsprk *Loop*[1]	✓	S	✓	✓	✓	✓	✓	E	✓	✓	✓	✓
Ttsprk *Split*[2]	✓	✓	✓	✓	✓	✓	✓	E	✓	✓	✓	✓

6.2.2 Comparison of Different S–ISP Assignment Algorithms

This section compares the static scratchpad assignment algorithms that were introduced in the Section 4.1 regarding their impact on the WCET estimate of the system. To evaluate the different algorithms the generated basic block and function assignments of all benchmarks for all memory sizes are used to obtain WCET estimates. The best performing assignment algorithms will be used as competitors to the dynamic memories in the following WCET evaluations. To ease the readability of this section the complete comparison for the different benchmarks is located in Appendix A. In the following only one benchmark per memory type is chosen to exemplify the behaviour of the algorithms for the basic-block-based and the function-based scratchpad assignment.

BBS–ISP

The ILP formulations to find an assignment of basic blocks to the BBS–ISP, as discussed in Section 4.1.3, use benefit terms for every basic block ($benefit_{(i,j)}$ for knapsack-based assignment respectively $cx_{(i,j)}$ for WCP-sensitive assignment). The benefits are calculated by the difference of the execution cost when the basic block is located in the off-chip memory and the cost if it is in the scratchpad. As shown in Section 5.2.2 the execution cost is determined by a timing model that takes the initial instruction window (IW) content of each basic block into account. As also discussed in Section 5.2.6 there are more precise ways to take the execution context of each basic block into account. Anyway, the basic block assignment is not aware of the execution



(a) Comparison of results obtained by knapsack-based basic block assignment

(b) Comparison of results obtained by WCP-sensitive basic block assignment

Figure 6.4: WCET estimates for Compress using BBS–ISP showing the normalised WCET estimates for assignments found with knapsack-based (KN) and WCP-sensitive (WS) approaches for different BBS–ISP sizes (The values are normalised to the optimal WCET estimate of both graphs. KN and WS doesn't use penalties for basic block assignment, KN* and WS* doesn't use penalties for basic block assignment and the cost for the basic blocks is not recalculated after assignment, KNJ and WSJ use penalties as shown in Table 6.11.)

context of the basic blocks, it considers only the benefit of putting the basic block in the very same execution context into the scratchpad.

Because the relocation of basic blocks might require additional instructions to reconnect the control flow, the basic block timing can be affected. The timing changes can be classified in two categories: First direct timing changes, like when adding an extra jump instruction to reach a moved basic block or a short jump is extended to a long jump causing an additional fetch to obtain the basic blocks code. The second category covers the indirect timing changes, which are caused by modifications of the basic block's execution context. For example the execution context of a basic block is changed, if it is entered by an additional connective jump instead by continuous addressing. This is the case because on any jump the CarCore processor flushes the IW, such that the basic block is entered with an empty IW causing additional fetch effort.

Due to these changes in the timing of the basic blocks, which are not modelled in the basic BBS–ISP assignment strategies (knapsack-based (KN) and WCP-sensitive (WS)), the changes of the basic block timing are underestimated. Therefore, these algorithms create non-optimal assignments that will not lead to the reduction of the WCET estimate that is expected. Even decisions may be made by the algorithms that increase the WCET estimate when the size of the BBS–ISP is also increased, which is against the assumption that with increasing BBS–ISP the WCET estimate will be continuously declined.

As shown in Figure 6.4 this occurs for the benchmark Compress. The WCET estimates for the KN and the WS show for some BBS–ISP sizes an increase compared to smaller BBS–ISP sizes. As discussed above, this is caused by omitting the effect of basic block timing changes

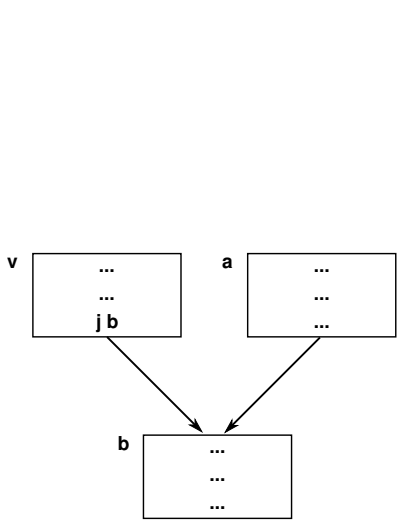
in the assignment algorithm. The estimates KN* and WS* deliver the view of the impact of the assignment algorithms, i.e. no basic block timing changes occur on assigning a basic block (beside the benefit of a block). Since the basic block time has to be recalculated to obtain a valid WCET estimate, the KN* and WS* provide only hypothetical (possibly underestimated) estimates. The comparison of these estimates with the WCET estimates obtained by KN and WS[20] shows the impact of the impreciseness, when the modified basic block context (e.g. by reconnection of the relocated basic blocks) is not taken into account. Furthermore, the KN and WS algorithms cannot reach optimal WCET estimates, even if the whole code could be copied into the BBS–ISP, which is the case for the largest BBS–ISP size in Figure 6.4. This is caused by the fact that a block that has no direct execution cost benefit will not be copied into the BBS–ISP, resulting in a rather large amount of timing penalties for connective jumps.

To circumvent the underestimation of basic block timing effects and deliver lower WCET estimates by better assignments, jump and size penalties where introduced for the basic block assignments. For estimating the jump penalties different jump scenarios are assumed, as described in Section 4.1.3 and quantified in Table 6.11. The results for these basic block assignment algorithms are shown as KNJ (knapsack-based assignment including jump and size penalties) and WSJ (WCP-sensitive assignment including jump and size penalties) in the Figure 6.4. The WCET estimates are lower than the estimates of the assignment algorithms that do not consider the basic block timing changes. This is caused by the maximisation of the benefit for all basic blocks and also taking the jump and size penalties into account. As shown in Figure 6.4 the consideration of the timing changes for the basic blocks delivers the (nearly) minimal WCET estimate for large BBS–ISP sizes, by also assigning basic blocks without a benefit to the BBS–ISP to minimise the penalty terms in the ILP formulations. The knapsack-based algorithm (KNJ) is not able to reach the minimum possible WCET (marked by the dashed line at 1.0 in Figure 6.4), because it is only able to assign basic blocks that are on the WCP and the WCP changes when assigning all basic blocks from the WCP. This occurs in the shown example for the function `cl_block` that is executed in a loop. Therefore, even small changes (as if on the WCPs only one basic block differs) of the WCP can significantly bias the overall WCET estimate. The WCP-sensitive approach is aware of changes of the WCP and thus can reach the minimal WCET as Figure 6.4(b) shows.

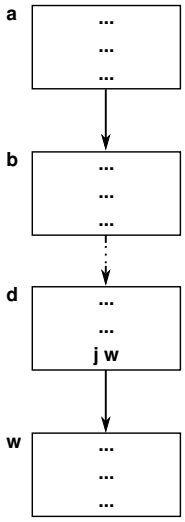
The usage of the jump scenarios for KNJ and WSJ (introduced in Section 4.1.3) provide only an upper bound of the penalties that occur on assigning certain blocks, which causes an overestimation of the basic block timing changes that might result in a non-optimal BBS–ISP assignment. This occurs for example in the WCET estimates shown in Figure 6.4 for KNJ at BBS–ISP size of 1,088 B (normalised size of ≈ 0.84) and all following sizes and for WSJ for size of 544 B (≈ 0.42) and 704 B (≈ 0.54). For these cases a set of basic blocks that were assigned for smaller BBS–ISP sizes is not assigned to the scratchpad, because the penalty of assigning them is overestimated. So another set of basic blocks is assigned, that is larger and now fits the BBS–ISP. In fact due to the overestimation of the penalties of basic blocks that were assigned before, the real benefit of the assigned basic blocks is lower than expected by the assignment algorithm. Thus a non-optimal decision is made by the assignment algorithm.

The overestimation of the penalties of some basic blocks is caused by the assumption that on a connecting jump the IW is flushed and has to be completely reloaded on the execution of the reconnected basic block. But this has not to be always the case, because the IW may already assumed to be empty by ISPTAP (due to multiple entry points of a basic block) or the loss of the IW content does not contribute in a penalty that requires the complete IW reload. For clarification both cases are briefly described in the following example.

[20] For KN and WS the same assignments are calculated as for KN* and WS*, but ISPTAP properly takes the timing changes of the basic blocks into account.



(a) Basic block b is entered by a and by a jump from v.



(b) Basic block b starts a path that is ended by an unconditional jump to w.

Figure 6.5: Scenarios for the overestimation of the jump penalty by BBS–ISP assignment algorithms

Example. Consider the basic blocks a and b that are connected without a jump. Furthermore, basic block a is assigned to the BBS–ISP, whereas b is not assigned. Then the jump penalty jp_{ca} from Table 6.11 of $3 + 3 \cdot MMAT$ cycles is considered by the KNJ and WSJ algorithm. This penalty is overestimated, if

1. b is also be entered by a jump from v (see Figure 6.5(a)). Then ISPTAP assumes an empty IW, because the different execution contexts when b is entered by a or v are not distinguished. So the penalty of filling the IW on entering of b is already contained in the execution cost of b. Hence, the penalty assumed by the assignment algorithm is overestimated by $3 \cdot MMAT$ cycles.

2. b starts a path that contains less instructions that fit the IW and the path ends with an unconditional jump to basic block w (see Figure 6.5(b)). Then the whole IW does not need to be fetched to execute the path beginning with b. So the jump penalty is overestimated by the number of cycles charged for fetch accesses that will not be needed to execute the path, which are $(3 - \text{size}_{fetch\ blocks}(< b, \cdots, d >)) \cdot MMAT$ cycles. If b is directly followed by w, the path length is 1. Further, the basic blocks of path $< b, \cdots, d >$ have to be in the same memory.

3. case 2 also holds, if the jump is conditional. Then the penalty is overestimated for the taken branch only.

4. case 2 also holds, if instead of a jump a function is called.

So these sources of overestimations of the jump penalties are caused by the execution context of the basic blocks that have to be reconnected on the assignment of one of them. Furthermore, as the second scenario describes the source of overestimation might be beyond the scope of the two basic blocks that are affected by the insertion of a connective jump. ♦

Another source of overestimating the penalty occurs, if one short jump or short call has to be extended to reach the block in the other memory. Therefore, a penalty of one additional fetch request is charged (refer to jp_{js} and jp_{cs} from Table 6.11). Depending on the alignment of the short branch instruction in the basic block, this penalty is necessary when executing the altered basic block or not. Since the jump penalties are fixed and independent of the characteristics of the basic blocks, the upper bound of one additional fetch request is assumed.

To mitigate the overestimation of the penalties in the basic block assignment for each pair of basic blocks an appropriate jump penalty has to be assigned. This customized penalty has to consider the difference of the exact execution contexts of two connected basic blocks for the case they are both in the same memory and for the case they are not. Furthermore, to have an exact jump penalty beyond the two basic blocks under consideration the neighbourhood of these blocks has also to be taken into account. The usage of customized jump penalties can deliver more optimal basic block assignments and thus may lead to lower WCET estimates. Also it can ensure the continuously declining of the WCET estimates, if the size of the BBS–ISP is increased. The implementation of the customized jump penalties is out of scope of this work, but it is seen a future perspective to improve the quality of the basic block assignment. The work on a WCP-sensitive basic block assignments published in [Falk and Lokuciejewski, 2010] also uses fixed jump penalties. Hence, the preciseness of BBS–ISP assignment algorithms with fixed penalties is considered as suitable for a comparison of the different instruction memories.

As shown in Figure 6.4 the WSJ algorithm is superior in finding the basic blocks with the highest impact on the WCET esimates on assignment to the BBS–ISP compared to the other algorithms. The normalised WCET estimates for all benchmarks of Table 6.12 using the different BBS–ISP assignment algorithms shown in Appendix A.1 corroborate the choice of WSJ. Therefore, the BBS–ISP with WSJ assignment algorithm is used in all following WCET evaluations as competitor to the dynamic memories.

FS–ISP

For the FS–ISP also a knapsack-based and a WCP-sensitive function assignment algorithm was discussed in Section 4.1.2. Both algorithms deliver the same function assignments, if the WCP does not change due to the assignment of the selected functions. But if the WCP changes the WCP-sensitive algorithm will provide a function assignment that results in a lower WCET estimate. As example the WCET estimates for the Sha benchmark are presented in Figure 6.6.

The figure shows that the WCET estimates of the WCP-sensitive algorithm (denoted as WS) are at least as good as the estimates the knapsack-based algorithm (denoted as KN) delivers. But for many memory sizes it provides a more optimal FS–ISP assignment resulting in lower WCET estimates. The reason for the better assignment of the WS algorithm is that the WCET path of Sha changes due to the assignment of some functions. The Sha algorithm used in the MiBench suite [Guthaus et al., 2001] is based on the secure hash standard published in [NIST, 1995]. The standard defines a set of different logical functions that are selected depending on the index of a surrounding loop. The Algorithm 6.1 shows how the appropriate function is selected. In the analysed version of the Sha benchmark (refer to the comment on the Sha benchmark in Section 6.2.1) `f1` to `f4` are implemented as functions, according to the macro definitions `f1` to `f4` of the original MiBench code.

In the WCET analysis of the benchmark it is discovered that the function `f3` will have the highest WCET compared to the other functions `f1`, `f2`, and `f4`. Therefore, `f3` is on the WCP and the knapsack-based function assignment algorithm has a benefit of putting this function into the FS–ISP. Since all other functions have no contribution to the WCP their benefit is 0. When

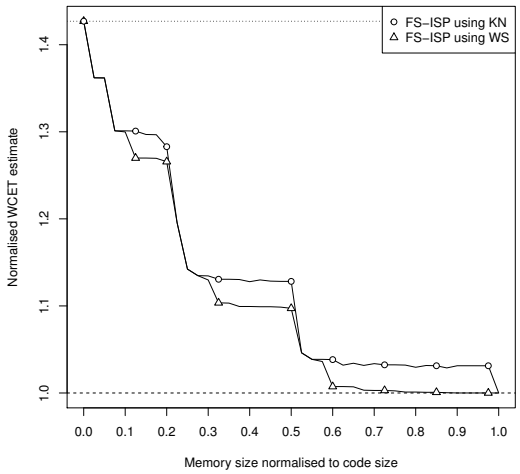


Figure 6.6: Comparison of WCET estimates of FS–ISP assignment algorithms for Sha (WCET estimates for knapsack-based (KN) and WCP-sensitive (WS) are normalised to optimal WCET estimate.)

putting `f3` into the FS–ISP the one of the other functions will be on the WCP[21]. So the KN algorithm cannot find the optimal function assignment. Anyhow, it tries to fill the FS–ISP to capacity and thus it also assigns functions with a benefit of 0 to the FS–ISP, if they are fitting and no function with a higher benefit can be assigned. Because the functions `f1` to `f4` are very small comparing to the code size (ranging from 20 to 26 B), there is likely some free memory space they can fit. Thus for example for the memory size of 352 B (≈ 0.28 the code size) the same assignment as the WS algorithm proposes is generated by the KN algorithm. So the KN algorithm can reach a lower WCET estimate by accident when assigning functions that are not on the WCP. Anyhow, the WS algorithm is capable to estimate the effects of WCP changes and so deliver optimal FS–ISP assignments. If the WCP does not change, the WS algorithm generates the same assignment as the KN algorithm. Hence, the WS algorithm is the algorithm of choice. Therefore, in the following this assignment algorithm is used for the comparison to the dynamic memories. The normalised WCET estimates for all benchmarks from Table 6.12 obtained by both FS–ISP assignment algorithms are provided in Appendix A.2.

Notice that both algorithms do not consider penalties caused by short call instructions, because it is supposed that the impact of penalties caused by the replacement of short calls by long calls is negligible, comparing to the size and benefits of a function.

6.2.3 Comparison of the D–ISP with Common Instruction Memories

In this section the WCET estimates for the D–ISP with FIFO replacement policy are compared the estimates of static scratchpads (BBS–ISP using WSJ and FS–ISP using WS) and a fully associative LRU cache. In all evaluations the interference penalty at the off-chip memory level is assumed for the static scratchpads and caches. So the MMATs of the last column from

[21] For `f1`, `f2` and `f4` the benefit of assigning them to the FS–ISP is equal. So it does not matter which of these functions is on the WCP after assigning `f3` to the FS–ISP.

Algorithm 6.1 Sha core algorithm derived from [NIST, 1995, see Section 5. FUNCTIONS]

Input: 32 bit words: B, C, D and index: t
Output: Value of Logical Function f_t

```
if 0 ≥ t ≥ 19 then
    f ← (B ∧ C) ∨ ((¬B) ∧ D) // Implemented as function f1(B, C, D)
else if 20 ≥ t ≥ 39 then
    f ← B ⊕ C ⊕ D // Implemented as function f2(B, C, D)
else if 40 ≥ t ≥ 59 then
    f ← (B ∧ C) ∨ (B ∧ D) ∨ (C ∧ D) // Implemented as function f3(B, C, D)
else if 60 ≥ t ≥ 79 then
    f ← B ⊕ C ⊕ D // Implemented as function f4(B, C, D)
end if
```

Table 6.9 (p. 173) are applied. Since the D–ISP does not suffer interference penalties the lower MMAT for off-chip memory access of 4 cycles is used. If the S–ISP memories are of the same size as the benchmark code, no interference at the level of the off-chip memory can appear. Thus, the absence of interferences can be assumed for this particular S–ISP size and the same off-chip MMATs as for the D–ISP can be used. For consistency reasons in the following figures this is not done and for all S–ISP sizes the same MMATs are assumed. Notice that the WCET estimate of an S–ISP of the benchmark's size will always be optimal. The comparison of the D–ISP to this optimal WCET estimate is discussed separately in the Section 6.2.4.

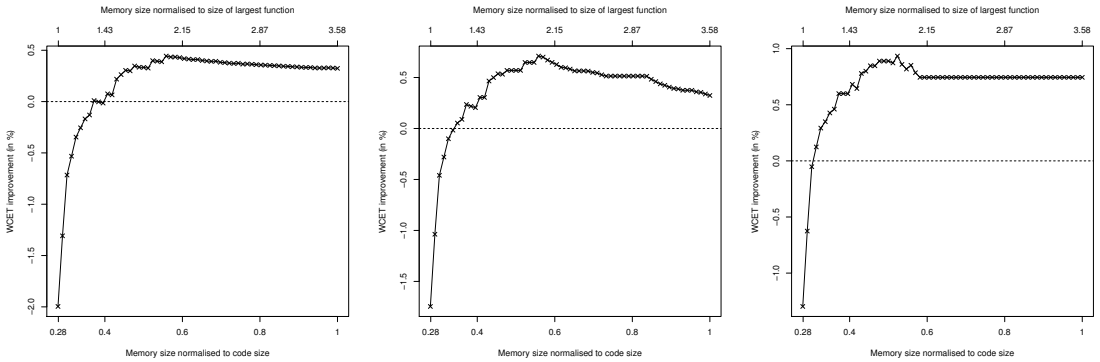
The Figures 6.7 to 6.19 show the improvement of the WCET estimate by the use of the D–ISP compared the BBS–ISP (WSJ), FS–ISP (WS), and the fully associative LRU cache of the same size. The *dashed* line at the WCET improvement of 0 represents the equality of the WCET estimates for both compared memories. If the difference plotted in the figures is positive, a lower WCET estimate is reached by the D–ISP. Otherwise the usage of the D–ISP results in a higher WCET estimate for the particular memory size. The presented figures only compare the WCET for memory sizes beginning at the size of the largest function and are increased in steps of 32 B until the size of the benchmark code is reached.

Two x-axes are shown in the Figures 6.7 to 6.19. The lower one shows the size of the memory normalised to the size of the benchmark code. The upper x-axis represents the memory size that is normalised to the size of the largest function. Notice that the size of the largest function in the benchmark according to Table 6.12 (p. 175) cannot be exactly matched by the memory size. This is due to the restriction that the internal block size of the D–ISP is 8 B and the cache line size is 32 B (refer to Table 6.9). Moreover, for all memories the same sizes are used in the evaluation, otherwise a direct comparison of the different WCET estimates would not be possible. Thus all memory sizes are multiples of 32 B. Hence, the normalised size of 1 at the upper x-axis represents the size of the largest function rounded up to the next multiple of 32 B.

In Appendix B the normalised WCET estimates for all S–ISP and cache sizes are depicted. The Figures B.1 to B.15 show the development of the WCET estimates for the different memories that are compared in this section with increasing size. Also the WCET estimates for the S–ISP and cache memories with sizes smaller than the largest function are presented. Furthermore, WCET estimates for the fully associative FIFO cache and the direct mapped cache are available in this appendix. Within these figures the normalised WCET of the D–ISP is also shown and they might be used as reference to the figures shown in this section, which present only the relative WCET improvements for the D–ISP. In addition to that the Table B.1 gives the WCET estimates in cycles for the two baseline configurations: on-chip memory only and no on-chip memory. The first was used to normalise the WCET estimates shown in the Appendix B.1.

Adpcm

The Figure 6.7 shows that the WCET estimates for Adpcm are similar for the static scratchpads, the LRU cache, and the D–ISP. Anyhow, the D–ISP reaches a small improvement of the estimates, which is below 1% compared to the three other memories. The figures also show that the D–ISP performs worse than all three memory types, when its size equals the size of the largest function. This is because of the increased effort needed by the D–ISP to handle evictions of the larger function, if a small one is loaded. The impact of the evictions is rather low, because the benchmark consists of two phases that are represented by two large functions (`encode` and `decode`), which call multiple rather small functions in a flat call hierarchy. Also some of the smaller functions contain loops, such that the function code is worth to be copied into the scratchpad.



(a) BBS–ISP (WSJ) vs. D–ISP (b) FS–ISP (WS) vs. D–ISP (c) I–Cache (LRU) vs. D–ISP

Figure 6.7: Adpcm WCET comparison for D–ISP vs. BBS–ISP, FS–ISP, and I–Cache

For all D–ISP sizes larger than 1.5 times the largest function the D–ISP performs better than the other memories. The decent improvement of the WCET of below 1% is caused by the fact that the most frequently executed parts of the benchmark is located in two small functions (`my_fabs` and `my_sin`). These functions fit into a static scratchpad of a size about one forth of the size of the largest function in the benchmark, which is approximately 0.07 the benchmark's code size. So with this tiny S–ISP nearly optimal WCET estimates are reachable, as Figure B.1 shows. The focus of the application on this small part also leads to a low LRU cache miss rate on the worst-case path, such that also the cache reaches superior estimates for small memory sizes. Furthermore, the D–ISP cannot profit from the absence of memory interferences since the amount of load/store instructions is quite low for this benchmark (as later shown in Table 6.16 of Section 6.2.5) and thus only a few situations occur on which these interferences have to be charged to the WCET estimates for the systems using S–ISPs or cache.

Compress

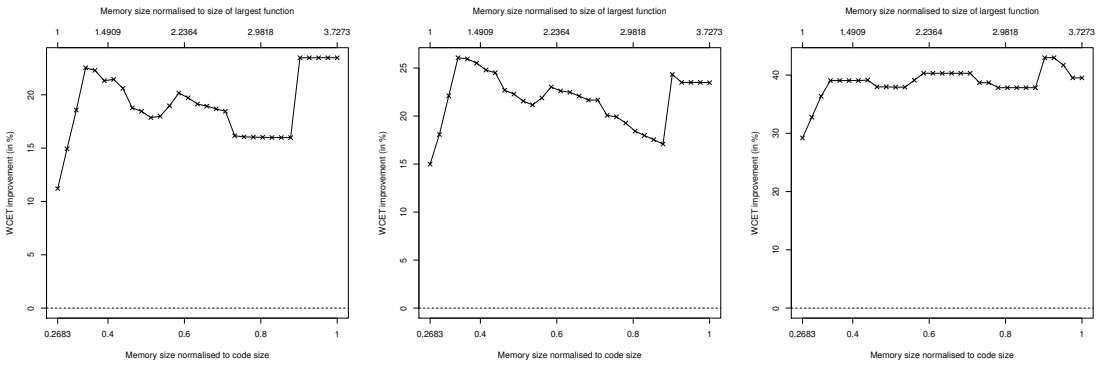
For Compress the D–ISP performs significantly better than the other memories. This holds also, if the D–ISP size is as large as the largest function. The Figures 6.8(a)-(c) show that it can reach a WCET estimate that is more than 20% lower than for BBS–ISP or FS–ISP and even about 43% lower than for a cache of the same size.

Notice that for example the decrease of the difference between the compared memory and the D–ISP does not necessarily result of a higher WCET estimate of the D–ISP it can also be the

result of a lower estimate of the cache or scratchpad, respectively. Nevertheless, it is also possible that the WCET estimate for the D–ISP increases, if its size is also increased. For example for the size of 608 B, which is about of 46% of the benchmarks code size, the WCET estimate is larger than for the smaller memory size of 576 B, as depicted in Figure B.2. The increase of the WCET estimate when having a larger memory is caused by the FIFO replacement policy of the D–ISP: Due to the larger scratchpad size a longer history of accessed functions can be maintained and the fact that recently accessed functions might be evicted next, leads to the effect that sure hits for a smaller memory size may be turned into "not classified" (NC) accesses by the analysis of a larger memory size. For a safe WCET estimation NC accesses are treated by the timing analysis similar as a misses and thus the function loading penalty is charged.

So the following case occurs of Compress: on call of the function `cl_block` it is not contained in any possible scratchpad state for the D–ISP size of 576 B, such that it has to be loaded. For the larger D–ISP size of 608 B there exist one memory state s in which `cl_block` is in the scratchpad from a previous execution. All other memory states do not contain `cl_block`. Therefore, the function call is categorised as NC. Nevertheless, the activation of the function has to be assumed to be a miss in the D–ISP. For both memory sizes (576 B and 608 B) the function is added to the first-in position of all D–ISP memory states, except for the state s (with memory size of 608 B). According to the FIFO policy the position of the accessed function is left unchanged in s. Unfortunately, the function `cl_block` gets evicted from s during execution of `cl_block`, by calling other functions. So the function is not contained in the successor state of s on reactivation of `cl_block` after returning from one of its callees. Since in all other memory states for size 576 B and 608 B the function remains in the D–ISP, the reactivation is classified as AH for the size of 576 B and NC for the size of 608 B. Thus, for the larger memory size the analysis has to charge the penalty to load the function `cl_block`, which leads to the higher WCET estimate. This effect is due to imprecisions of the analysis, because no valid concrete execution path exists in which the function `cl_block` has to be loaded on call and return (for the path that traverses s it is loaded only on return and for all other paths is loaded only on call).

The Figure B.2 also show that the D–ISP performs better than the other memories of any size. This is caused by the absence of the interference penalties at the D–ISP and the high fraction of load/store instructions on the WCP of the application that are more costly for a static scratchpad and the cache (according to Table 6.9). However, the impact of the interference penalty will be discussed separately in Section 6.2.5.



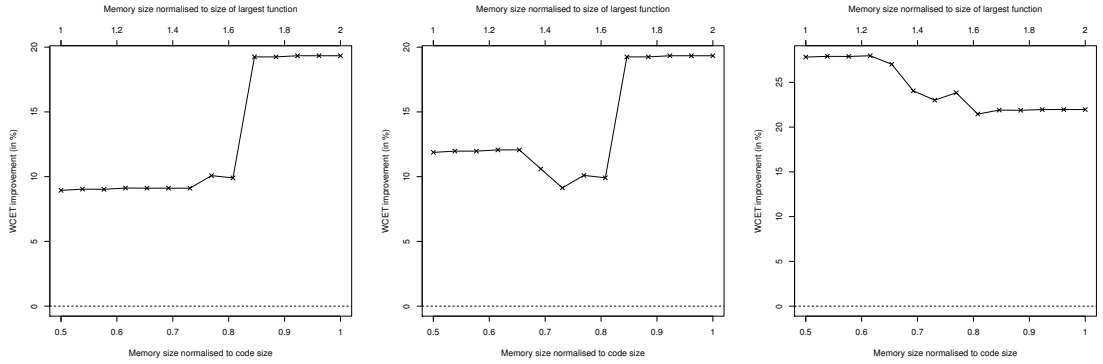
(a) BBS–ISP (WSJ) vs. D–ISP (b) FS–ISP (WS) vs. D–ISP (c) I–Cache (LRU) vs. D–ISP

Figure 6.8: Compress WCET comparison for D–ISP vs. BBS–ISP, FS–ISP, and I–Cache

Dijkstra

As for Compress also for Dijkstra the D–ISP performs better than every other memory, independent of their size. The reduction of the WCET estimates ranges from 21% to 27% for the LRU cache and from 9% to 19% for the static memories. The differences of Figure 6.9(a) and 6.9(b) until the memory size of 0.75 the code are caused by the capability of the BBS–ISP to assign the code of the critical path on the granularity of basic blocks. Therefore, the BBS–ISP already reaches a close to optimal WCET estimate for the size of 0.4 of the code size. For larger BBS–ISP sizes only basic blocks with minor impact on the WCET are added into the BBS–ISP (cf. Figure B.3). The FS–ISP needs a larger memory size to fit all functions that have a high contribution to the worst-case critical path, such that the close to optimal WCET is reached not below a memory size of 0.75 of the code size. The largest contribution to the estimated WCET for the FS–ISPs is when the `enqueue` function can be assigned to the memory, which is possible at about 0.1 of the code size. This function contains a loop that dominates the WCP, because the basic blocks of the loop contribute about 83% to the WCP. Hence, the execution of the function `enqueue` determines the critical part of the benchmark.

In Figure 6.9(c) it can be seen that the difference of the WCETs estimated for the cache and the D–ISP are reduced with larger memory sizes. Figure B.3 can clarify this effect: the WCET estimates for the cache are improved until the size of 0.7 of the code size, whereas the D–ISP estimates stick nearly at the same level until the memory size of about 0.85 of the code size.



(a) BBS–ISP (WSJ) vs. D–ISP (b) FS–ISP (WS) vs. D–ISP (c) I–Cache (LRU) vs. D–ISP

Figure 6.9: Dijkstra WCET comparison for D–ISP vs. BBS–ISP, FS–ISP, and I–Cache

The D–ISP reaches already at its minimal size a superior WCET estimate, because the function `enqueue` and all other functions it calls fit the scratchpad. If all functions that are called in the main loop of the benchmark can be maintained concurrently in the D–ISP, the D–ISP reaches its lowest estimated WCET. This is possible with a memory size of about 0.85 of the benchmark's code size, which is also shown in the Figure B.3 in the Appendix B.

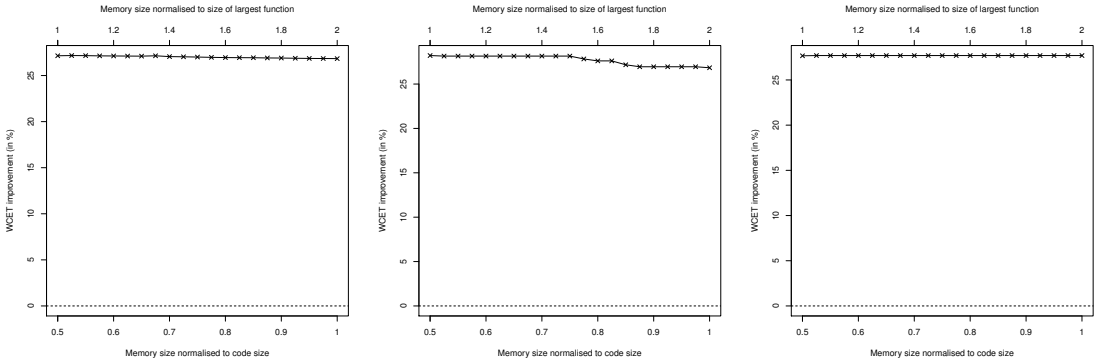
If just a part of the frequently accessed functions fit the D–ISP, no further significant WCET improvements compared to the minimal D–ISP size can be noticed. This is because within every loop iteration each called function and the function that contains the loop has to be evicted once, caused by the FIFO replacement policy that always evicts the oldest function. The stack-based replacement policy can here benefit by keeping the caller function in the memory, as Figure 6.21 shows, which is discussed in detail in the evaluation of the different D–ISP replacement policies in Section 6.2.6.

Edn

The comparison of the D–ISP to the scratchpads and the cache shown in Figure 6.10 depicts a nearly constant reduced WCET for the D–ISP of about 27%. Only for the FS–ISP comparison there is a noticeable reduction of the difference from 28% to slightly below 27%. This is caused by the fact that the last two functions that have a noticeable contribution on WCET-critical path can only be assigned, if the memory size is slightly smaller than the code size. Notice that the WCET estimates for the S–ISP and cache are already at a close to optimal value for the minimal D–ISP size. As shown in Figure B.4 the major contribution on the WCET estimate is reached already with memory sizes of about 0.3 of the code size for the static scratchpads and 0.4 of the code size for the cache. Thus at any larger memory size only minor changes of the WCET estimate arise.

Also the WCET estimate of the D–ISP is constant for any memory size. This is caused by the flat call hierarchy of the benchmark and the fact that only one larger function is contained in the benchmark. The benchmark function calls all other functions sequentially and only once, such that a function call can never be a hit in the D–ISP. Hence, only on return to the benchmark function the D–ISP can profit when keeping the benchmark function in the scratchpad. By the fact that the largest function of the benchmark is about half the size of the whole benchmark code, nearly the rest of the benchmark fits the D–ISP on its smallest possible size. Thus for the D–ISP with minimal size only two misses on returning to the benchmark function occur: the first is caused by its eviction on call of the largest function and the second occurs after the sequential call of a nearly all other functions from the benchmark function (which are slightly larger then the D–ISP size).

One of the two misses on returning to the benchmark function can be saved, if the D–ISP is as large as the code minus the largest function. To get rid of the second miss the D–ISP has to be at least of the same as the whole code. The impact of saving these misses is not noticeable in the Figures 6.10 and B.4, because the benchmark function is small and the penalty of loading it adds only a minor increase on the WCET (below 1‰ of the overall WCET).



(a) BBS–ISP (WSJ) vs. D–ISP (b) FS–ISP (WS) vs. D–ISP (c) I–Cache (LRU) vs. D–ISP

Figure 6.10: Edn WCET comparison for D–ISP vs. BBS–ISP, FS–ISP, and I–Cache

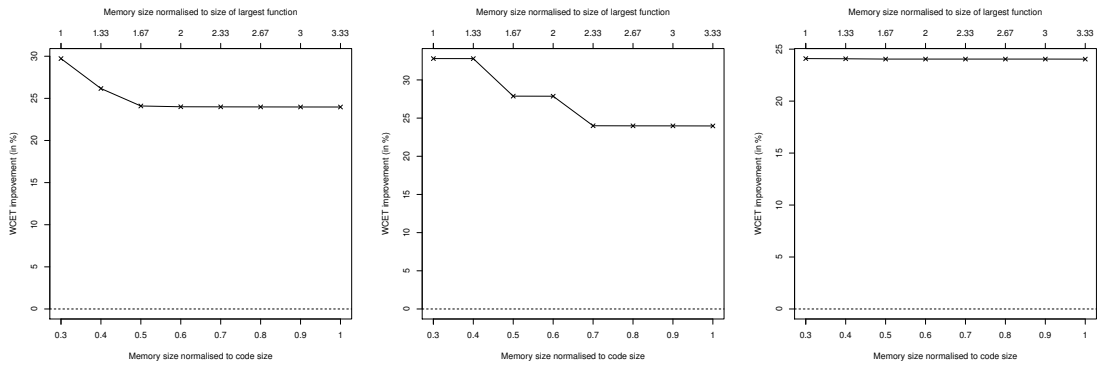
The *Loop* configuration of Edn in which the main benchmark function is executed in a loop shows a very similar WCET behaviour. Therefore, the comparison figure is omitted in this section. The content of the D–ISP can be reused, if the functions called within a loop. But due to the flat call hierarchy, this can happen only for the D–ISP size that is as large as the code. Furthermore, the impact of reloading the benchmark function is nearly negligible. This is because

of the low cost of reloading the functions compared to their execution time. For completeness the normalised WCET estimates for Edn *Loop* are shown in Figure B.5 of Appendix B.

Matmult

For the Matmult benchmark shown in Figure 6.11 a 30% to 24% lower WCET estimate is reachable with the D–ISP compared to S–ISP and cache memories of the same size. The difference of the BBS–ISP and the FS–ISP is again caused by the capability of the BBS–ISP to select the critical parts of the application more precisely and assign them to the scratchpad. Thus the BBS–ISP reaches its near to optimal WCET estimate already at a size of 0.5 of the code size, whereas the FS–ISP reaches the same WCET estimates not below the size of 0.7 of the code size. For a more detailed view on the WCET estimates of the S–ISPs, refer to Figure B.6 that shows the development of the WCET estimates for increasing memory sizes.

The Figure 6.11(c) shows a constant difference of the WCET estimates of the D–ISP compared to the LRU cache. This is because the WCET for the cache reaches a minimal value at the size of the largest function and the WCET for the D–ISP is also constant at any memory size. This behaviour for the D–ISP caused by the effect that while the `call`/`return` microcode is executed the D–ISP already starts fetching the code of the activated function. Because the microcodes take more than 50 cycles to execute (refer to Table 5.1, p. 142) the D–ISP can fetch about 92 B and activate the function within this time, according to the MMAT shown in Table 6.9 (p. 173), the fetch width of 64 bit, and the activation time on miss shown in Table 6.10. So since the functions in Matmult are very small, all functions, except the largest function, can be loaded into the D–ISP without any penalty. The largest function of Matmult has a miss penalty, but due to the fact that it is a leaf of the call tree and it is not called within a loop, the miss occurs only once on call of the function and cannot be avoided. So independently of hit or miss for the other functions this leads to the constant WCET estimate for any D–ISP size. Furthermore, as shown in Figure B.6 the WCET estimate of the D–ISP is always lower than the estimates for every size of the other memories.



(a) BBS–ISP (WSJ) vs. D–ISP (b) FS–ISP (WS) vs. D–ISP (c) I–Cache (LRU) vs. D–ISP

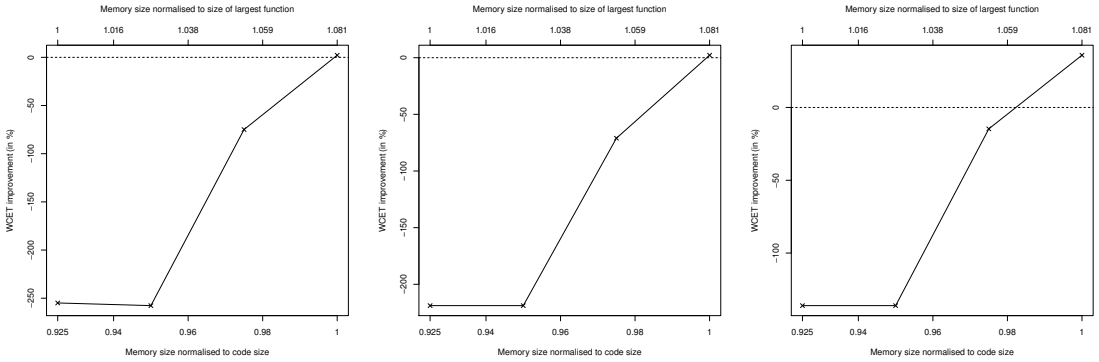
Figure 6.11: Matmult WCET comparison for D–ISP vs. BBS–ISP, FS–ISP, and I–Cache

The Matmult *Loop* configuration has the same WCET behaviour for all analysed memories as Figure B.7 shows. This is because the D–ISP benefits from the function load in the slipstream of the microcode execution and the minor impact of the penalty of loading the largest function multiple times within the main loop. For that reason the figures comparing the D–ISP with each memory separately for the Matmult *Loop* benchmark are not shown.

Puwmod

Comparing the WCET estimates of the D–ISP with the ones of the other memories for Puwmod shows that the D–ISP performs worst. This is caused by the very large benchmark function that calls frequently small functions (in sum 17 call points), which evict each other frequently. For example for every function call and return the activated function has to be reloaded, if the D–ISP size equals the size of the largest function. This results in 35 misses[22] in the scratchpad per benchmark run. Unfortunately, the Puwmod benchmark function does not contain any loops, thus each instruction loaded into the D–ISP (and also into the caches) is executed only once per benchmark run. Furthermore, the code of the benchmark function contains three times the same code, such that the frequent eviction of this function has a high impact on the WCET estimate of the application. Therefore, the D–ISP has an up to 3.55 times higher WCET estimate than a BBS–ISP, 3.19 times higher compared to the FS–ISP, and 2.36 times higher than a cache of the for its minimal size as shown in Figure 6.12.

If the D–ISP size is increased to 1,248 B which is one 32 B step before the benchmarks code size is reached, the number of misses is reduced to 12. This leads to a noticeable reduction of the overhead needed by the D–ISP. But the D–ISP reaches only comparable estimates, if it has the size of the benchmarks code and each function misses only once. It allows a 2% lower WCET estimate than the S–ISP memories and a 36% lower WCET estimate compared to the cache. Due to the fact that no code of the benchmark function is reused, the D–ISP and the cache have to load each instruction into the memory before it is accessed once. Therefore, both memories perform even worse than using no on-chip memory, since the miss penalty of the cache and the D–ISP is higher than a simple off-chip memory access. This is depicted by Figure B.8. Despite that, the D–ISP finally reaches a similar estimated WCET performance as the S–ISPs, mainly because of its capability to load parts of functions in the slipstream of the microcode execution and the absence of interference penalties.



(a) BBS–ISP (WSJ) vs. D–ISP (b) FS–ISP (WS) vs. D–ISP (c) I–Cache (LRU) vs. D–ISP

Figure 6.12: Puwmod WCET comparison for D–ISP vs. BBS–ISP, FS–ISP, and I–Cache

To reduce the size of the largest function in the benchmark and to introduce locality that the dynamic memories can exploit the main benchmark function was split such that each of the three iterations is executed by a function called with different parameters. Because the code of one iteration has to be encapsulated to distinguish the different execution contexts and due

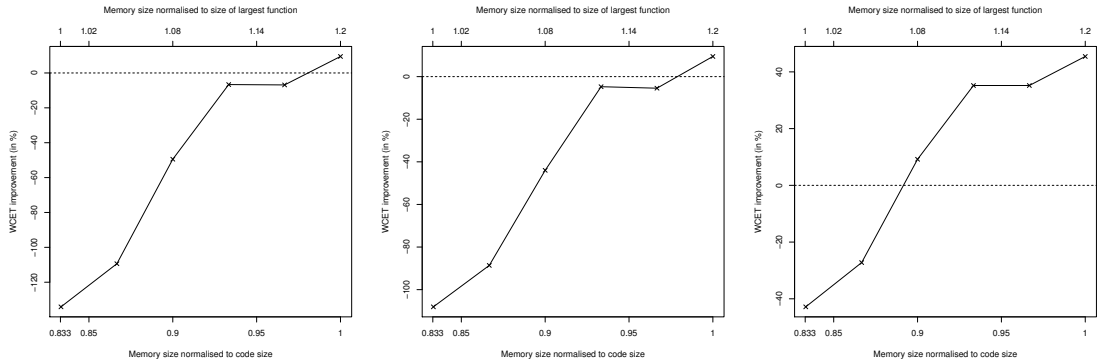
[22] 17 misses for calling one of the 2 callee functions, 17 to reload the benchmark function on returning to it, and one miss for the first activation of the benchmark function.

to the call of the additional function, the overall run-time of the benchmark is increased when using no dynamic memory. For Puwmod the WCET estimate of the *Split* configuration is about 15% higher than the WCET estimate of the original benchmark, when considering the on-chip memory configuration only[23]. The comparison of the WCET estimates for Puwmod *Split* is shown in Figure 6.13 and the normalised WCET values are shown in Figure B.9.

Comparing Figure 6.13 with Figure 6.12 for the unchanged Puwmod benchmark, it can be seen that the WCET estimates for D–ISP are improved. Nevertheless, for the minimal D–ISP size the WCET estimate for equally sized memories is still 2.34 times higher than for the BBS–ISP, 2.08 times for the FS–ISP, and 1.42 times for the cache. The D–ISP performs better for the *Split* configuration, because the benchmark function is smaller and it is reused in by each iteration. In the original benchmark always the code of all three iterations has to be loaded on D–ISP miss, whereas in the *Split* configuration only the code of one iterations is to be copied into the D–ISP. Thus less unused instructions are loaded. Anyhow, the frequently reloading of all functions leads to an estimated performance that is much worse than the performance of the other memories.

The performance drawback is relaxed, if one complete iteration (represented by executing the split benchmark function and all its callees) can be executed without D–ISP miss. This can be seen for the D–ISP size of 0.93 of the code size in Figure 6.13. From that size the D–ISP performs better than the cache of same size and it can reach a lower WCET estimate than a system using no on-chip memory as depicted in Figure B.8. Using a D–ISP that is as large as the code it can reach a 9.5% improved WCET compared to the S–ISP memories and a 43% lower WCET when comparing it to the cache.

As for the unchanged benchmark the cache does not reach a lower WCET estimate than a system that uses only the off-chip memory. This is caused either by the low amount of code reuse (The benchmark does not contain any loop.) or by the low spatial code locality (The code contains many control flow paths with only few instructions, such that it is possible that just a part of code in the cache lines is used.). Thus the D–ISP is able to outperform the cache memory.



(a) BBS–ISP (WSJ) vs. D–ISP (b) FS–ISP (WS) vs. D–ISP (c) I–Cache (LRU) vs. D–ISP

Figure 6.13: Puwmod *Split* WCET comparison for D–ISP vs. BBS–ISP, FS–ISP, and I–Cache

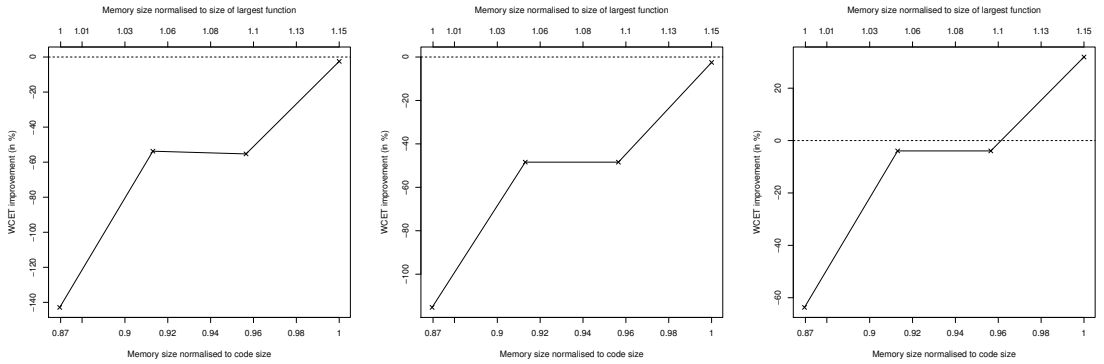
Rspeed

The structure of the Rspeed benchmark is very similar to the Puwmod benchmark, because both are EEMBC benchmarks. In Rspeed the largest function has a smaller contribution to the code

[23]Refer to the baselines for a system with on-chip memory only in Table B.1.

size (86.5% instead of 92.5% according to Table 6.12) in difference to Puwmod and it has only 8 calls to the two smaller function in the code of the benchmark function. So for the D–ISP of the size of the largest function all function activations (on call and return) are misses. This leads to a 2.43 times larger WCET estimate comparing to the BBS–ISP, 2.16 times comparing to the FS–ISP, and 1.64 times larger comparing to the fully associative LRU cache. The lower WCET overhead compared to Puwmod can be explained by the fewer function call points and thus the reduced number of misses in the D–ISP for its smallest possible size. Due to in Rspeed 8 times a function is called the maximum number of misses[24] is 17. The development of the WCET difference for Rspeed is shown for the different compared memories in Figure 6.14.

As for Puwmod competitive WCET estimates for the D–ISP compared to the S–ISP memories are only reachable, if the whole benchmark code fits the D–ISP. Nevertheless, the estimated WCET for the D–ISP is still about 2.5% higher than the WCET for both S–ISP memory. Compared to the cache a 32% lower WCET is reachable for the D–ISP of the size of the benchmark. As Figure B.10 shows any of the dynamic memories (all cache flavours and the D–ISP) cannot compete with the static scratchpads, because the code reuse is also very limited in this benchmark. The cache doesn't even reach a lower WCET estimate then if no on-chip memory would be used, as Figure B.10(b) shows. For the D–ISP an improvement compared to the use of no on-chip memory is only reachable for a D–ISP of the benchmark's size, as the figure also depicts.



(a) BBS–ISP (WSJ) vs. D–ISP (b) FS–ISP (WS) vs. D–ISP (c) I–Cache (LRU) vs. D–ISP

Figure 6.14: Rspeed WCET comparison for D–ISP vs. BBS–ISP, FS–ISP, and I–Cache

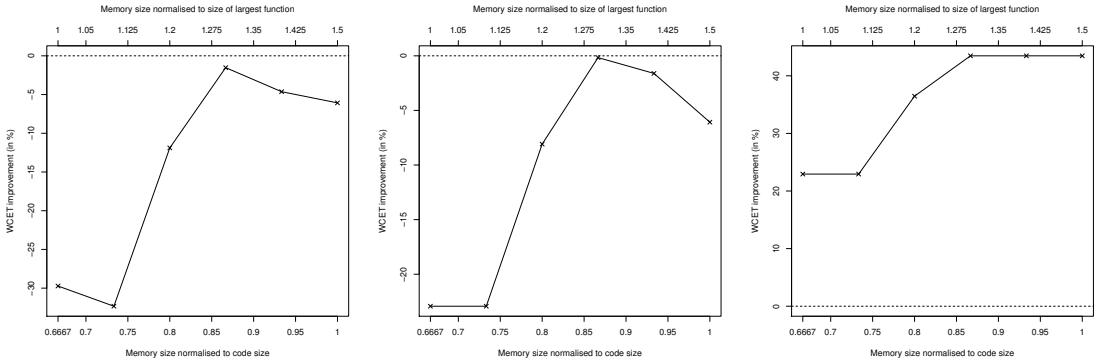
When splitting the large benchmark function the estimated D–ISP performance is improved compared to the other memories. This is depicted in the Figure 6.15. Notice that due to the code overhead to distinguish the different iterations within the created function and the additional function calls the WCET estimate for the baseline system for the *Split* configuration is somewhat increased (refer to Table B.1).

Due to the reduced frequently reloading of non-executed code for Rspeed *Split* compared to Rspeed the WCET overhead of the D–ISP for the size of the largest function is reduced to 1.3 times for the BBS–ISP, 1.23 times for the FS–ISP, and to an improvement of a 23% lower WCET estimate for the cache of the same size. The outlying value showing a worse WCET ratio of a 1.32 times higher estimate comparing the D–ISP with the BBS–ISP is caused by an improvement for the BBS–ISP, whereas the estimate of the D–ISP does not change. This is also

[24] 8 misses for calling one of the 2 callee functions, 8 misses on return to the benchmark function, since the loading of the callee function has evicted the caller, and one miss at call of the benchmark function that has to be initially loaded.

shown in Figure B.11(a). Compared to the S–ISP memories the D–ISP is able to provide WCET estimates that are in the same order of magnitude, but it requires a larger memory size to reach the same estimates as the Figure B.11(a) also depicts. Anyhow, the D–ISP cannot outperform the S–ISP memories, but it is able to reach an overhead lower than 10%.

In comparison to the cache the D–ISP always performs better and reaches even with its smallest possible size a lower WCET estimate than a cache of any size, which is shown in Figure B.11(b). This is because the cache never reaches a WCET estimate that is lower than the estimated WCET of a system without any on-chip memory. In contrast to the cache the D–ISP provides lower WCET estimates comparing to a system without on-chip memory with any D–ISP memory size.



(a) BBS–ISP (WSJ) vs. D–ISP (b) FS–ISP (WS) vs. D–ISP (c) I–Cache (LRU) vs. D–ISP

Figure 6.15: Rspeed *Split* WCET comparison for D–ISP vs. BBS–ISP, FS–ISP, and I–Cache

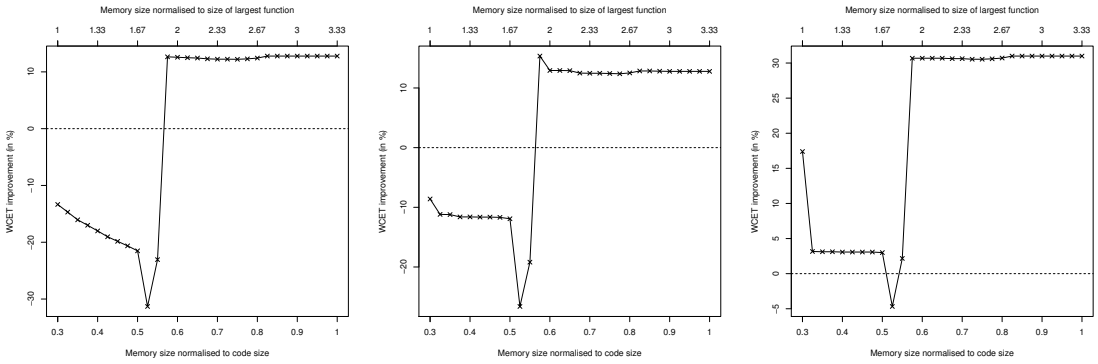
For Rspeed *Split* the optimal WCET estimate is not reachable for the D–ISP at size of the benchmark. This is because of the internal fragmentation of the different functions in the scratchpad: the needed scratchpad size for the whole benchmark code is 488 B instead of 474 B, which is the size of the benchmark. So a slightly lower WCET estimate for the D–ISP is reachable, if whole code can really fit the scratchpad. However, the figures do not show any values beyond the size of the benchmark.

Sha

For Sha the S–ISP memories have superior WCET estimates for small memory sizes as Figure 6.16 illustrates. So the D–ISP cannot compete for small memory sizes. Its estimated WCET is ranging from 1.13 times the WCET of the BBS–ISP for the smallest possible D–ISP size to 1.31 times the WCET of the BBS–ISP in the worst case. Comparing to the FS–ISP the gap is smaller and ranges from 1.09 times the WCET with a D–ISP of the size of the largest function to 1.27 times the WCET in the worst case. The difference of the BBS–ISP and FS–ISP is again caused by the coarser granularity of the entities the FS–ISP can assign. After a noticeable performance jump at a memory size of about 0.6 the code size the D–ISP reaches an up to 12.5% lower estimated WCET compared to both S–ISP memories.

The Figure 6.16(c) shows that the D–ISP can always outperform the cache in terms of a lower WCET, except for one outlying WCET estimate for one D–ISP memory size. Ignoring this single value the D–ISP can reach a 2% to 31% lower WCET than a cache of the same size. The relative reduction of the D–ISP improvement compared to the other memories for memory

sizes up to 0.5 of the code size shown in Figure 6.16 is caused by stagnating WCET estimates of the D–ISP. Refer to Figure B.12 for the development of the normalised WCET estimates of all memory types.



(a) BBS–ISP (WSJ) vs. D–ISP (b) FS–ISP (WS) vs. D–ISP (c) I–Cache (LRU) vs. D–ISP

Figure 6.16: Sha WCET comparison for D–ISP vs. BBS–ISP, FS–ISP, and I–Cache

The Figure 6.16 shows an negative outlier for the D–ISP comparing to all other memories at the size of slightly above 0.5 of the code size. This is caused by a WCET increase for the D–ISP, which is also depicted in Figure B.12. This anomaly shows that an increasing D–ISP size does not necessarily lead to a lower WCET estimate. The reason for this is similar as for Compress. In detail the following case occurs: On call of the function `FUNC` it is categorised as miss for the D–ISP size of 640 B. For the size of 672 B (for which the anomaly occurs) it cannot be categorised as hit or miss, because one part of the possible memory states contains the function and the other part does not. Hence, the call is handled as NC and it is assumed that the function has to be loaded. The function `FUNC` calls one of the functions `f1` to `f4` (shown in Algorithm 6.1, p. 183). So in the analysis for both memory sizes for each possible control path one of these functions has to be loaded. At size 672 B there exists at least one memory state in which the function `FUNC` is already contained on its call, otherwise the call would not be classified as NC. For such concrete memory state the load of one of the functions `f1` to `f4` evicts the caller function `FUNC`. Therefore, on return to the function `FUNC` it cannot be classified as sure hit for the D–ISP size of 672 B. Whereas at D–ISP size of 640 B the return to `FUNC` will always be a hit. Since the analysis has to charge the loading penalty when a function is not classified as hit, a higher WCET is calculated for a D–ISP with the larger size.

When looking at the source of this effect, it is clear that the higher WCET estimate is caused by imprecisions of the memory analysis. During the analysis every function activation that cannot be properly categorised as hit or miss is charged with the penalty to load the function. But if every possible memory state is examined separately, it shows up that the memory state that evicts the function `FUNC` on return does not need to load it on call and on the other hand and the memory state that does not evict the function `FUNC` on return has to load it on call. So for each program path the function has to be loaded only once. By the abstraction that all possible memory states are treated as a set the constraints for the particular memory states are blurred, causing that a higher loading penalty is charged than possible. Nevertheless, due to the fact that the penalty is overestimated the analysis is safe and does not underestimate the WCET. Because the function `FUNC` is executed frequently in loops, this effect appears as a noticeable impact on the WCET.

The next interesting point the Figures 6.16 and B.12 show is the reduction of the WCET estimate to about 30% from one memory size to the next step at a D–ISP size of 736 B (57.5% of the code size). The reason for the abrupt WCET reduction is that the function that contains the hot spot loops of the benchmark and all its callees fit the D–ISP, such that during the execution of these loops no evictions in the D–ISP take place.

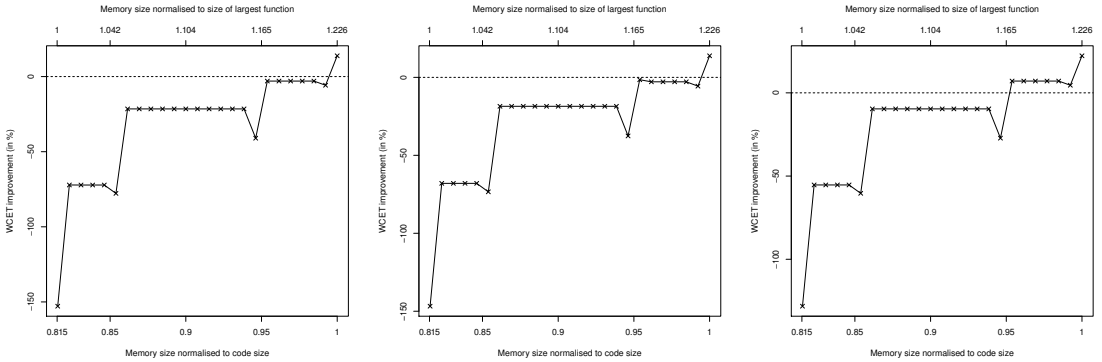
Any further increase of the D–ISP size beyond 2 times the size of the largest function does not significantly decrease the WCET. Thus this D–ISP size can be seen as optimal for the Sha benchmark. Furthermore, the D–ISP can benefit of loading small functions for free, because it can exploit the time the call and return microcodes need to process by starting the load of the functions. This affects especially the tiny functions that were converted from defines into functions (refer to Section 6.2.1). So the D–ISP is able to reach superior WCET estimates compared to the other memories.

Ttsprk

The Figure 6.17 shows that the D–ISP hardly reaches similar WCET estimates as the static scratchpads for Ttsprk. In the worst case the usage of the D–ISP results in a 2.53 or 2.47 times higher WCET estimate than for a BBS–ISP or a FS–ISP of the same size, respectively. The D–ISP is only able to outperform the S–ISP memories for the case that the complete application fits the D–ISP. As Figure B.13 shows the BBS–ISP already reaches a close to optimal WCET on a size of 0.1 of the code size. For the FS–ISP the nearly optimal WCET estimate is reached at about 0.15 of the code size. This is because the main contribution to WCP is caused by the two functions `YTableLookup` and `ZTableLookup`, which are called in sum from 27 points in the main benchmark functions. `YTableLookup` contains one loop and `ZTableLookup` contains two loops, whereas the rest of the code is mainly sequential. All three loops are rather small and fit a scratchpad size of 128 B. If these loops are assigned to the BBS–ISP, the WCET estimate can only be further decreased by about 10% for any larger BBS–ISP size. For the FS–ISP both functions have to be assigned to reach an nearly optimal estimate, which is the case for the FS–ISP size of 576 B (about 0.14 of the code size).

Due to the hot spot of the application is located in the small loops of the `YTableLookup` and `ZTableLookup`, the cache can also reach good WCET estimates at small memory sizes, which is also depicted in the Figure B.13. Anyhow, the cache cannot compete with static scratchpads, because there are large code parts that are executed only once, e.g. the largest function of the benchmark. This is because the Ttsprk benchmark is an EEMBC benchmark, in which the code in the main benchmark function is replicated (see also the benchmarks Puwmod and Rspeed). Therefore, the cache loads many instructions it uses only once. Compared to the D–ISP the cache provides lower WCET estimates until the main benchmark function (which is also the largest one) and the two table lookup functions (`YTableLookup` and `ZTableLookup`) fit the D–ISP. So the D–ISP, which has with lower sizes an up to 2.28 times higher WCET estimate, can reach a 22% lower WCET estimate than the cache.

The Figure B.13 shows four steps at which the WCET estimate of the D–ISP is significantly reduced. The first step is the next memory size after the initial size and it is caused by the fitting of the smallest function and the largest function into the D–ISP. The second step at about 0.85 the applications code size reduces the WCET estimate by about 30%. This is due to the fitting of `YTableLookup` and the benchmark function to the D–ISP. Because `YTableLookup` is called up to 15 times in the main benchmark function and in some cases it is called multiple times consecutively without calling any other functions in between, the D–ISP can benefit from maintaining only these two functions at once. Then the estimated WCET does not change until the two lookup functions and the benchmark function fit the D–ISP (except for the anomaly for



(a) BBS–ISP (WSJ) vs. D–ISP (b) FS–ISP (WS) vs. D–ISP (c) I–Cache (LRU) vs. D–ISP

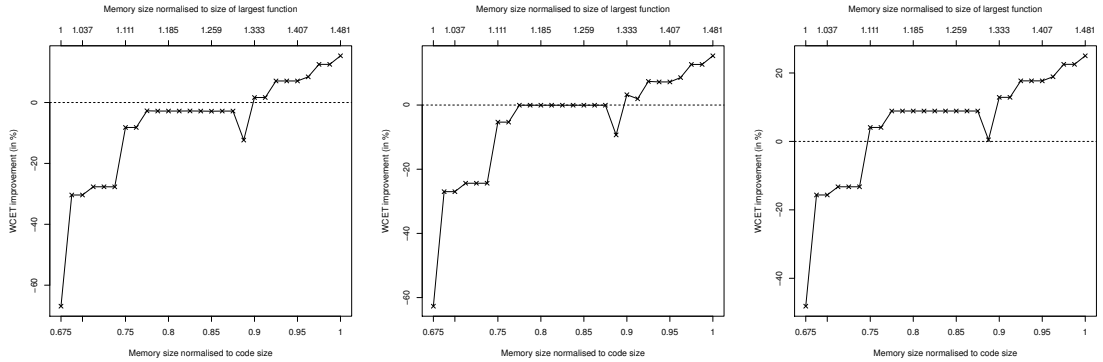
Figure 6.17: Ttsprk WCET comparison for D–ISP vs. BBS–ISP, FS–ISP, and I–Cache

the next smaller memory size at about 0.95 of the code size). Holding the benchmark function, `YTableLookup`, and `ZTableLookup` is very beneficial, because the functions are called in the same part of the application frequently. So maintaining these three functions the D–ISP does not have to evict or load a function until another function is called. The reason why there is no change of the estimated WCET if the larger `ZTableLookup` function and the main benchmark function both match the D–ISP, is that `ZTableLookup` is never called twice without any interrupting call of another function. This causes the eviction of `ZTableLookup` and the benchmark function before `ZTableLookup` is called again. The last performance jump is when all functions fit the D–ISP and each has to be loaded only once and never gets evicted.

Before each of these steps at which the WCET is reduced a WCET increase is noticeable in Figure B.13. These outliers are again caused by the fact that a sure miss at a smaller memory size could not be categorised as miss at a larger size on function call. Hence, on return to this function a sure hit at the smaller memory size is converted into a memory access that is neither classified as hit nor miss (NC) for the larger memory. This causes that the load penalty is charged twice for the affected function, first on its call and second when a callee returns to it, instead of only once as for the smaller memory size.

As for the other EEMBC benchmarks the main function of Ttsprk is split to reduce the minimal D–ISP size, which is bounded by the largest function in the code, and increase code reusing. It is depicted in Figure 6.18 that by this the D–ISP can compete with the other memories. For the smallest possible D–ISP size the D–ISP causes an overhead of about 67% and 63% on the WCET estimate of a BBS–ISP and a FS–ISP of the same size, respectively. But this overhead is halved for the next memory size step. With increasing memory sizes the D–ISP can reach a WCET improvement of about 15% compared to the S–ISP memories. Compared to the cache the estimated D–ISP performance is much better for the *Split* version of Ttsprk. Starting from a WCET overhead of 48% that is lowered to 16% by the next memory size step the D–ISP finally reaches a 25% lower WCET estimate than the cache.

The development of the WCET estimates of Ttsprk *Split* shown in B.15 is similar to the original version of the benchmark. The WCET estimate for the D–ISP is significantly decreased, if the `YTableLookup` and the split benchmark function fits. Due to the changed call graph in which a function calls the split benchmark function three times, the reduction of the estimated WCET for the D–ISP is not as large as for the unchanged Ttsprk benchmark. Furthermore, when looking at the WCET estimates of the Ttsprk *Split* benchmark just one noticeable outlier

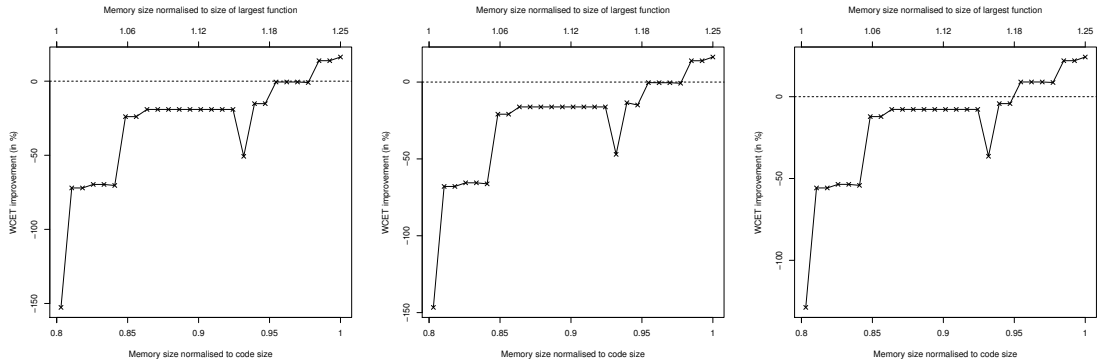


(a) BBS–ISP (WSJ) vs. D–ISP (b) FS–ISP (WS) vs. D–ISP (c) I–Cache (LRU) vs. D–ISP

Figure 6.18: Ttsprk *Split* WCET comparison for D–ISP vs. BBS–ISP, FS–ISP, and I–Cache

for the D–ISP can be found. It occurs at the D–ISP size of slightly below 0.9 of the code size and is also caused by the NC classification of function calls and returns, which are caused by inaccuracies in the representation of the memory states during analysis.

In Table 6.12 (p. 175) also a *Loop* configuration of the Ttsprk benchmark is listed. As the WCET estimates are very likely to the estimates of the unchanged Ttsprk configuration the Figures 6.19 and B.14 will not be discussed in detail. But notice that for both configurations (and also the *Split* configuration) the development of the WCET estimate with increasing memory sizes has the same shape for the D–ISP. This is caused by the characteristics of the benchmark in which two functions called by one benchmark function dominate the worst-case behaviour of the benchmark.



(a) BBS–ISP (WSJ) vs. D–ISP (b) FS–ISP (WS) vs. D–ISP (c) I–Cache (LRU) vs. D–ISP

Figure 6.19: Ttsprk *Loop* WCET comparison for D–ISP vs. BBS–ISP, FS–ISP, and I–Cache

Conclusion

The close look on the different benchmarks shows that the D–ISP is able to outperform static scratchpads and instruction caches, if the critical parts of the application can fit the D–ISP. This is e.g. the case for a loop in which several functions are called: only if the frequent reloading

of functions in the loop body is avoided by an appropriate D–ISP size, the D–ISP can provide lower WCET estimates than other memories. A drastic improvement of the estimated WCET that can arise at the memory size in which the D–ISP is able to hold the critical part of the application. This is exemplified best by the Sha benchmark.

For benchmarks in which the largest function dominates the whole application, as for the EEMBC benchmarks, the D–ISP has to be close to the size of the application to reach WCET estimates comparable to S–ISPs and caches. But when splitting the largest function and thereby also increasing the code reuse for the EEMBC benchmarks, the WCET estimates for the D–ISP can be significantly improved as the *Split* configurations show. Anyhow, it is supposed that in safety critical systems the size of functions should be rather small to improve readability and ease the verification of the code, as recommended by Holzmann [2006]. Benchmarks with smaller functions like the applications from the Mälardalen suite or the MiBench suite allow the D–ISP to outperform the competing memories at sizes between 1 to 2 times the size of the largest function.

Furthermore, the evaluation shows that the cache performs slightly worse than the S–ISP memories for the considered memory sizes. There are two sources of this observation: first every instruction that is cached has to be copied into the cache before it can be used, whereas the content of the scratchpads is already fixed at run-time. Second, the cache accesses that cannot be classified are assumed by the analysis to be a miss. This pessimism and the additional effort to copy the instructions result in higher WCET estimates. But the Figures B.1 to B.15 also show that the cache can outperform the S–ISPs for small memory sizes for some benchmarks. So the cache can also deliver outstanding WCET estimates especially, if it can make use of the better utilisation of the given memory space by its dynamic content management.

If the temporal locality of the application code is low (e.g. because it only consists of sequential code without any loops) the dynamic memories cannot really compete with the static scratchpads, as the EEMBC benchmarks Puwmod and Rspeed show. For these benchmarks the cache cannot even perform better in the worst case than a system having no on-chip memory, which is depicted in the Figures B.8(b) and B.10(b). This is due to the cache needs to load the cache line before it can deliver the requested memory reference, which is more costly than just obtaining the memory reference directly from the off-chip memory[25]. So if the memory references are only used once, a system with cache is slower than one without. This also holds for the D–ISP that also has to load the functions before they can be executed. So the static scratchpads have the advantage that the assigned code is already contained in the memory on application start. Therefore, if the dynamic memories cannot benefit from their content management, their performance will be lower than for a S–ISP memory.

The Table 6.14 shows the average improvements of the WCET estimates for the D–ISP compared to the S–ISPs and the LRU cache of the same size. In the first column the memory size normalised to the size of the largest function is shown. The columns two to four contain the average WCET improvement for the BBS–ISP (WSJ), FS–ISP (WS), and the I–Cache (LRU) for all benchmarks shown in the Figures 6.7 to 6.19 except the *Loop* configurations and the unchanged EEMBC benchmarks (Puwmod, Rspeed, and Ttsprk). The number of benchmarks for which the estimated average WCET improvement is calculated in Table 6.14 declines with increasing memory size, because all benchmarks have a different ratio of the largest function to the code size. The number of benchmarks used to calculate the mean average is shown in the fifth column of the table. Since the memory size steps in Table 6.14 are normalised to the size of the largest function in the benchmark and the estimation of the WCETs considers only memory size steps of 32 B (according to the description in Section 6.2.1), for some benchmarks

[25] One cycle for hit/miss detection has always to be added onto the cost for loading a line. Furthermore, it is possible that not all instructions in the cache line will be needed resulting in an extra overhead.

Table 6.14: Average WCET improvement of the D–ISP compared to BBS–ISP (WSJ), FS–ISP (WS), and I–Cache (LRU) for Adpcm, Compress, Dijkstra, Edn, Matmult, Puwmod *Split*, Rspeed *Split*, Sha, and Ttsprk *Split*

Size Norm. to Largest Function	**Average WCET Improvement**			**Number of Benchmarks**
	BBS–ISP	**FS–ISP**	**I–Cache**	
1.0	-18.8%	-12.9%	6.3%	9
1.1	-1.4%	2.0%	17.4%	9
1.2	6.7%	9.4%	23.4%	9
1.3	7.7%	10.8%	21.6%	8
1.4	8.5%	11.4%	22.5%	8
1.5	8.9%	11.2%	23.2%	8
1.6	10.3%	13.3%	19.4%	6
1.7	11.3%	14.3%	19.2%	6
1.8	10.3%	12.3%	18.5%	6
1.9	16.9%	18.6%	23.9%	6
2.0	16.9%	18.1%	23.9%	6
2.2	14.2%	15.4%	23.9%	4
2.4	13.9%	14.9%	23.9%	4
2.6	13.8%	14.6%	23.9%	4
2.8	13.3%	14.3%	23.6%	4
3.0	13.3%	13.9%	23.4%	4
3.2	13.3%	13.7%	23.4%	4
3.4	11.9%	12.4%	21.9%	2
3.6	11.9%	11.9%	20.1%	2

Size Norm. to Code Size	**Average WCET Improvement**			**Number of Benchmarks**
	BBS–ISP	**FS–ISP**	**I–Cache**	
1.0	13.9%		28.8%	9
	-5.0%			

and memory size steps in the table a WCET estimate is not available. Thus the missing values are approximated by the mean average of surrounding values.

The table shows that the D–ISP can reach up to 17% lower WCET estimates than an equally-sized BBS–ISP, even slightly better WCET estimates compared to a FS–ISP, and an improvement of up to 24% is possible comparing to an LRU cache. Notice that all competing memories support smaller sizes than the D–ISP, but for most benchmarks there exists a memory size at which the D–ISP outperforms the other instruction memories.

At the bottom of Table 6.14 an average of the WCET improvement for the memories that are as large as the code size is presented. It shows the best possible performance for the dynamic memories, because every instruction needs to be loaded into the I–Cache and D–ISP only once. So the best possible WCET estimate of the LRU cache can be undermatched by the D–ISP by about 29% in average over all benchmarks. Notice that this is mainly due to the absence of the interferences at the off-chip memory connection in a system with D–ISP. The impact of the interference penalty will be discussed separately in Section 6.2.5.

Comparing the best estimated WCET performance of the S–ISPs, meaning the whole code is assigned to the static scratchpads, with the D–ISP shows that the D–ISP is able to reach an about 14% lower average WCET. The D–ISP is only able to outperform the S–ISP, because it is assumed that the load/store requests may interfere with the instruction fetches on the off-chip

memory level when the S–ISP is used. Due to all fetches are directed to the S–ISP at a memory size as large as the code, no memory access can interfere with another. So the lower number in the second column shows the case if the S–ISP does not have to consider the interference penalty. Then for all benchmarks the average WCET estimate for the D–ISP is only slightly higher than for the S–ISP (as Table 6.14 shows the WCET improvement is -5%). This overhead is due to the D–ISP has to load the code from off-chip memory before it can be executed. The comparison of the D–ISP to the optimal performance when using a code-sized S–ISP is shown in the Section 6.2.4 in more detail.

To find explanations for D–ISP boosts or cutbacks in the WCET estimates it is necessary to look into the details of the D–ISP content analysis. The programmer is capable to keep track of the D–ISP states during the data flow analysis, because the content of the D–ISP changes only on call and return and the D–ISP contains only functions (to which the programmer is familiar with). This allows the programmer to find the cause for performance bottlenecks and have an idea of possible actions to improve the WCET estimate by altering the D–ISP size or the characteristics of the functions (e.g. call frequency or position in call graph). For example for the EEMBC benchmarks the *Split* configurations were build to optimise the WCET estimates by reducing the size of the largest function. In contrast to the D–ISP analysis the understanding of the content changes for a cache is nearly impossible for larger applications, because the memory references (i.e. cache lines) are too numerous and abstract for the programmer to handle. A toolchain support for the D–ISP and caches may mitigate this and provide hints for the programmer to reduce the WCET estimate. However, it is easier for the programmer to have hints on the level of functions instead of cache lines. So a tool support for visualising the results of the memory analysis in the source code is worth an investigation, but out of scope of this work.

6.2.4 WCET Overhead of the D–ISP Content Management

In this Section the overhead of the D–ISP with FIFO replacement policy is compared to the optimal case, which is a static scratchpad that contains the complete code segment (i.e. the whole application code). Because in such memory organisation every fetch access is directed to the on-chip memory, no interferences of instruction and data memory access can arise at off-chip memory level, independently if the off-chip memory connection is shared or not. Therewith, the WCET analysis can assume the same MMATs for S–ISP and off-chip memory accesses as for a system with D–ISP (refer to Table 6.9 for the used number of cycles). The system with all instructions located in the on-chip memory is further denoted as optimal system

The D–ISP cannot reach the same WCET estimates as the optimal system, because it needs to copy each function at least once into the function scratchpad memory. Therefore, the minimal overhead is determined by the size of the code that the D–ISP has to load. If the code loaded into the D–ISP is often reused and does not contain too much dead code (e.g. parts of functions that are not reachable or are not reached by the WCP), the overhead of the D–ISP will be rather low.

The Table 6.15 classifies the WCET overhead of the D–ISP in different sizes compared to the optimal system. Therewith, the table quantifies the cost of the D–ISP's dynamic content management, that needs to load the functions before they are used. The D–ISP sizes in the table are normalised to the size of the largest function in the benchmark. The value 1.0 denotes the minimal possible D–ISP size. As memory size steps the same as in Table 6.14 are used. The size "max" denotes the D–ISP size in which the complete benchmark fits. If no exact value exist for the memory size of Table 6.15 it is extrapolated by the mean average of the surrounding WCET estimates. The WCET overhead shown in the table corresponds to the normalised WCET estimates of D–ISP shown in the Figures B.1 to B.15, because the WCET estimates in these

Table 6.15: Estimated WCET overhead of the D–ISP compared to the optimal system (For the optimal system all instructions are located in the on-chip memory and no interferences at the off-chip memory level can occur.)

	Memory Size Normalised to the Size of Largest Function									
Bench-	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**	**1.6**	**1.7**	**1.8**	**1.9**
mark	**2.0**	**2.2**	**2.4**	**2.6**	**2.8**	**3.0**	**3.2**	**3.4**	**3.6**	**max**
Adpcm	2.9%	1.4%	1.0%	0.8%	0.7%	0.6%	0.4%	0.3%	0.3%	0.2%
	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	**0.1%**
Compress	41.3%	34.2%	27.0%	20.0%	20.0%	20.0%	19.8%	22.1%	22.1%	22.1%
	22.1%	17.4%	17.4%	17.4%	19.8%	19.8%	19.8%	19.8%	9.1%	**9.1%**
Dijkstra	12.9%	12.8%	12.7%	12.7%	12.7%	12.1%	11.7%	0.1%	0.1%	≈0%
	≈0%	–	–	–	–	–	–	–	–	**≈0%**
Edn	0.6%	0.5%	0.5%	0.5%	0.5%	0.5%	0.5%	0.5%	0.5%	0.5%
	0.5%	–	–	–	–	–	–	–	–	**0.5%**
Matmult	≈0%	≈0%	≈0%	≈0%	≈0%	≈0%	≈0%	≈0%	≈0%	≈0%
	≈0%	≈0%	≈0%	≈0%	≈0%	≈0%	≈0%	–	–	**≈0%**
Puwmod	336%	65.2%	–	–	–	–	–	–	–	–
	–	–	–	–	–	–	–	–	–	**18.5%**
Puwmod *Split*	189%	57.4%	10.4%	–	–	–	–	–	–	–
	–	–	–	–	–	–	–	–	–	**10.4%**
Rspeed	190%	84.3%	–	–	–	–	–	–	–	–
	–	–	–	–	–	–	–	–	–	**20.8%**
Rspeed *Split*	64.8%	64.8%	35.9%	20.9%	20.9%	20.9%	–	–	–	–
	–	–	–	–	–	–	–	–	–	**20.9%**
Sha	40.7%	40.7%	40.7%	40.7%	40.7%	40.7%	40.7%	40.8%	46.9%	0.6%
	0.6%	0.6%	0.7%	0.6%	≈0%	≈0%	≈0%	–	–	**≈0%**
Ttsprk	211%	49.6%	32.6%	–	–	–	–	–	–	–
	–	–	–	–	–	–	–	–	–	**6.1%**
Ttsprk *Split*	106%	44.3%	26.7%	27.9%	14.2%	6.0%	–	–	–	–
	–	–	–	–	–	–	–	–	–	**4.3%**

figures are also normalised to the optimal WCET estimate. The minimal overhead is shown in Table 6.15 in the cells with the "max" memory size, since the minimal overhead occurs at the maximum D–ISP size, which is any size larger than the size of the code. With a D–ISP of maximal size each function needs to be loaded only once, resulting in minimal costs caused by the dynamic content management.

As shown in the table different benchmarks reach already a rather low WCET overhead at the normalised D–ISP size of 1.0. This is either due to the critical part of the application already fits the D–ISP (e.g. for Adpcm) or the loading of functions into the D–ISP is for free, due to overlapping of function load and call/return microcode execution (as for Matmult). A high overhead can be recognised for the EEMBC benchmarks, because of their low code locality, the high amount of dead code, and also the numerous function evictions during execution. As discussed earlier in Section 6.2.3 the disappointing D–ISP performance is improved by the splitting the main benchmark function (see Puwmod *Split*, Rspeed *Split*, and Ttsprk *Split*).

The values of the WCET overhead give an overview how much instructions need to be loaded by the content management of the D–ISP. In general the lower the overhead is the better is the estimated WCET performance. But no threshold can be found, for that the D–ISP outperforms the competing memories. For example for Compress the D–ISP provides superior WCET estimates even at the smallest memory size, but it has an overhead of above 40%. On the other hand for Rspeed *Split* the overhead reaches a value of slightly above 20%, but the D–ISP is has still a higher WCET than the S–ISP memories. Thus the overhead does not correspond to the estimated WCET performance compared to the other memories. This is because the performance of the other memories is also affected by the application's structure and the interference penalty that is charged to every access to the off-chip memory. Therefore, in the following section the impact of the interference penalty is investigated using the example of the LRU cache.

6.2.5 Impact of the Off-Chip Memory Connection on WCET Estimates

In this section it will be shown that the D–ISP can exploit the two-phased execution scheme to reach lower WCET estimates than other memories in a system with a shared off-chip memory connection. These benefits are caused by the absence of the interference penalties for the D–ISP, whereas for the competing memories the WCET analysis has to consider the penalty for a possible interference at the off-chip memory level on every off-chip memory access, i.e. every load, store, and fetch that is not directed to an on-chip memory. To quantify the impact of the interference penalty the benchmarks were analysed for the fully associative LRU instruction cache with the interference penalty (as already done for all previous evaluations) and without considering the penalty. Therefore, the two different MMATs of Table 6.9 (p. 173) were used. The lower MMAT represents the system without interferences, i.e. a system with separated off-chip memory connection (denoted as SEMC), whereas the higher MMAT assumes an interference at every off-chip memory access, i.e. a system with shared off-chip memory connection (denoted as SHMC). For the characteristics of the different memory hierarchies SHMC and SEMC refer to Figure 2.2(b) and 2.2(c) (see Section 2.7), respectively.

The Table 6.16 shows the estimated mean WCET improvement for the different benchmarks that is reached, if the interference penalty has not to be charged for a system with LRU cache (Avg. ΔWCET I$ SHMC vs. SEMC). Therefore, the average mean is calculated from all cache sizes beginning with 32 B up to the size of the benchmark in steps of 32 B. So for Adpcm the average WCET estimate is 1.3% lower, if no interference penalty need to be taken into account for a system with LRU cache. This draws two conclusions: (1) the benefit of a system with separated off-chip memory connection (SEMC) is a 1.3% lower WCET estimate comparing to a system with shared off-chip memory connection (SHMC) and (2) at most 1.3% of the WCET estimate of the system with shared off-chip memory connection (SHMC) are caused by the interferences at the off-chip memory level.

To get the source of the higher WCET estimates caused by the interference penalty the classification of the executed instructions on the worst case path (WCP) is also shown in Table 6.16. The instructions are classified as *Arithmetic*, *Branch* (including Calls and Returns) and *Load/Stores*. A high fraction of load/store instructions leads to a higher impact of the interference penalty, because load and store instructions access the off-chip memory and the interference penalty is to be charged by ISPTAP. To get an impression of how many instruction fetches will access the off-chip memory the average miss rate of the LRU cache executing the WCP is also shown (Avg. I$ MR on WCP). For the average value shown in the Table 6.16 cache sizes from 32 B to the benchmark's size in steps of 32 B are considered.

The table shows that what is expected: Benchmarks with a high fraction of off-chip memory accesses in terms of load/stores or instruction cache misses are very sensitive to the interference

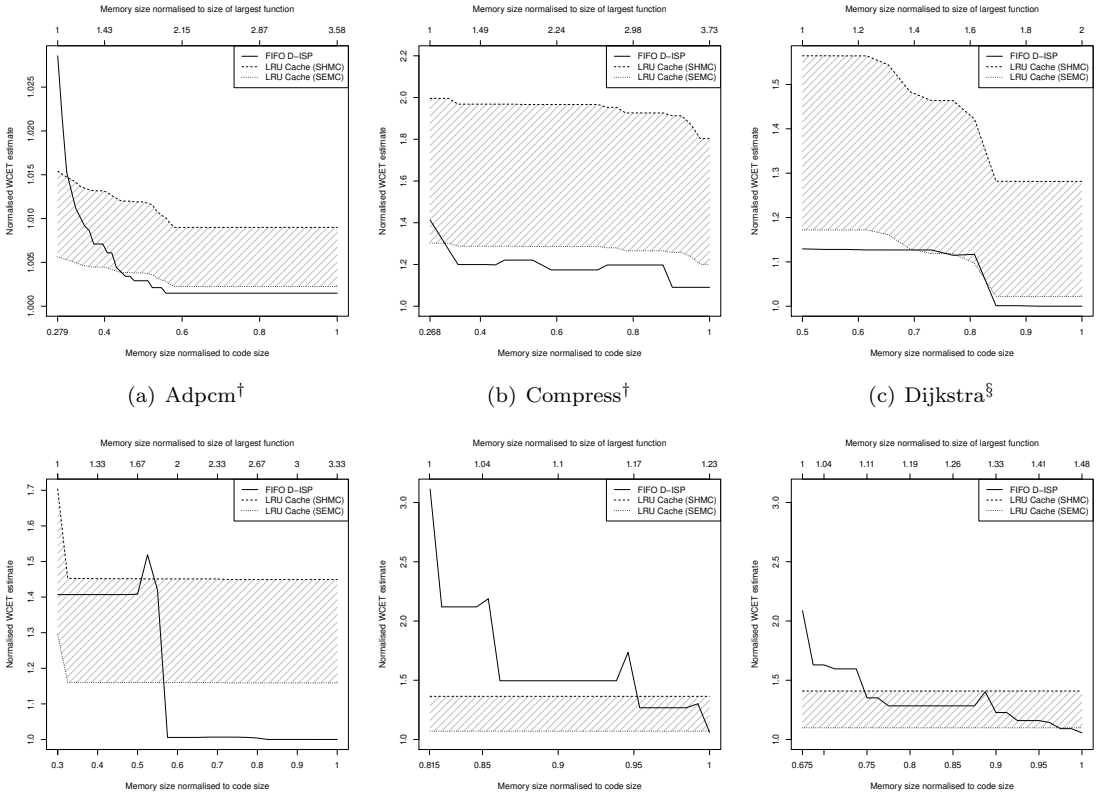
Table 6.16: Impact of interferences at the off-chip memory level for the LRU cache (The table shows the instruction classes on worst case path (WCP), average LRU cache miss rate (MR) on WCP, average WCET improvement if no interference penalty need to be charged (I$ SHMC vs. SEMC), and average WCET improvement of the FIFO D–ISP comparing to the LRU cache. The baseline for the last to columns is the LRU cache with interference penalties, i.e. I$ SHMC.)

Benchmark	**Instruction Classes on WCP**			**Avg. I$ MR on WCP**	**Avg. ΔWCET I$ SHMC vs. SEMC**	**Avg. ΔWCET I$ SHMC vs. D–ISP**
	Arith-metic	**Branches (w. C/R)**	**Load/ Stores**			
Adpcm	84.9%	14.3%	0.9%	1.8%	+1.3%	+0.7%
Compress	32.9%	11.0%	56.2%	29.8%	+34.8%	+38.7%
Dijkstra	50.2%	23.3%	26.5%	10.3%	+24.2%	+24.3%
Edn	65.4%	9.1%	25.5%	3.7%	+28.2%	+27.7%
Edn *Loop*	65.4%	9.1%	25.5%	3.7%	+28.1%	+27.7%
Matmult	59.2%	9.7%	31.1%	11.1%	+25.9%	+24.1%
Matmult *Loop*	59.2%	9.7%	31.1%	11.1%	+25.7%	+24.1%
Puwmod	42.1%	19.3%	38.6%	30.8%	+28.0%	-62.8%
Puwmod *Split*	50.3%	11.3%	38.4%	53.2%	+29.8%	+9.2%
Rspeed	60.2%	10.5%	29.3%	47.8%	+27.3%	-9.9%
Rspeed *Split*	62.1%	12.9%	25.0%	65.4%	+29.0%	+35.5%
Sha	59.8%	13.3%	26.9%	30.9%	+21.7%	+20.5%
Ttsprk	48.4%	29.3%	22.3%	4.0%	+22.0%	-19.2%
Ttsprk *Loop*	48.4%	29.3%	22.3%	3.9%	+21.9%	-16.2%
Ttsprk *Split*	48.3%	29.2%	22.5%	4.8%	+22.6%	+5.1%

penalty. So if many load/stores occur on the WCP as for Compress or if the cache miss rate on the WCP is high as for Rspeed *Split*, the interference penalty has a significant impact on the overall WCET. If no interferences are present, lower WCET estimates are possible, e.g. an average improvement of 34.8% for Compress and 29.0% for Rspeed *Split*. If a benchmark mainly consist of arithmetic instructions and poses only rare cache misses, the WCET estimate is hardly affected by the interference penalty, as Adpcm shows.

How much the D–ISP can benefit from the overhead caused by the interference penalty of the cache is shown in the last column of Table 6.16 (Avg. ΔWCET I$ SHMC vs. D–ISP). It quantifies the average WCET improvement of the D–ISP comparing to the LRU cache (assuming the interferences) for the memory sizes from the largest function up to the size of the application. The used WCET estimates for the average WCET improvements are already shown in the figures of Section 6.2.3. Notice, that the calculation of the average means for the cache miss rate (Avg. I$ MR on WCP) and the impact of the interference penalty for the cache (Avg. ΔWCET I$ SHMC vs. SEMC) contains also memory sizes smaller than the size of the largest function. For the original values of the impact of the interferences for the cache and the comparison of the cache and the D–ISP refer to the Tables B.2 to B.16 in Appendix B.2.

The comparison of the average D–ISP improvement (I$ SHMC vs. D–ISP) and the impact of the interference penalty for the LRU cache (I$ SHMC vs. SEMC) shows that the D–ISP can exploit the absence of the interferences to reach a WCET estimate reduction for most benchmarks that is in the same order of magnitude as the impact of the interference penalty for the cache. This is not the case for the EEMBC benchmarks for reasons of temporal locality of the code and thrashing of the D–ISP's FIFO managed content. In the case of Compress the D–ISP can



(a) Adpcm† (b) Compress† (c) Dijkstra§

(d) Sha§ (e) Ttsprk‡ (f) Ttsprk *Split*

Figure 6.20: Comparison of the WCET estimates of D–ISP and LRU cache with different off-chip memory connections (For the cache a shared off-chip memory connection (SHMC, according to Figure 2.2(b)) and a separated off-chip memory connection (SEMC, according to Figure 2.2(c)) configuration shows the impact of the chosen memory connection. To allow a better comparison the estimates of the Ttsprk and Ttsprk *Split* the WCET estimates of both benchmarks are normalised to the same estimate.)

even outperform a cache for which no interference penalties are taken into account, shown by a higher average WCET improvement for the D–ISP than the average difference of the WCET for the cache with and without considering interferences in Table 6.16. This is remarkable, because the D–ISP has always to load complete functions, independent of the path that is taken within the function. Whereas the cache can benefit from its fine-grained content management, if benchmarks contain long disjoint control flows and unused code parts in functions, as for example the original configurations of the EEMBC benchmarks.

The Figure 6.20 gives an insight of the WCET estimates for the D–ISP and the LRU cache with and without interferences for selected benchmarks[26]. The shown values are normalised to the WCET estimate of a system in which the whole code is located in a scratchpad and thus no memory interferences can occur. Notice that the graphs begin with the size of the largest function in the code, which is the minimal possible size for the D–ISP. In the figures the memory

[26]The data for complete set of benchmarks according to Table 6.12 is provided in Appendix B.2.

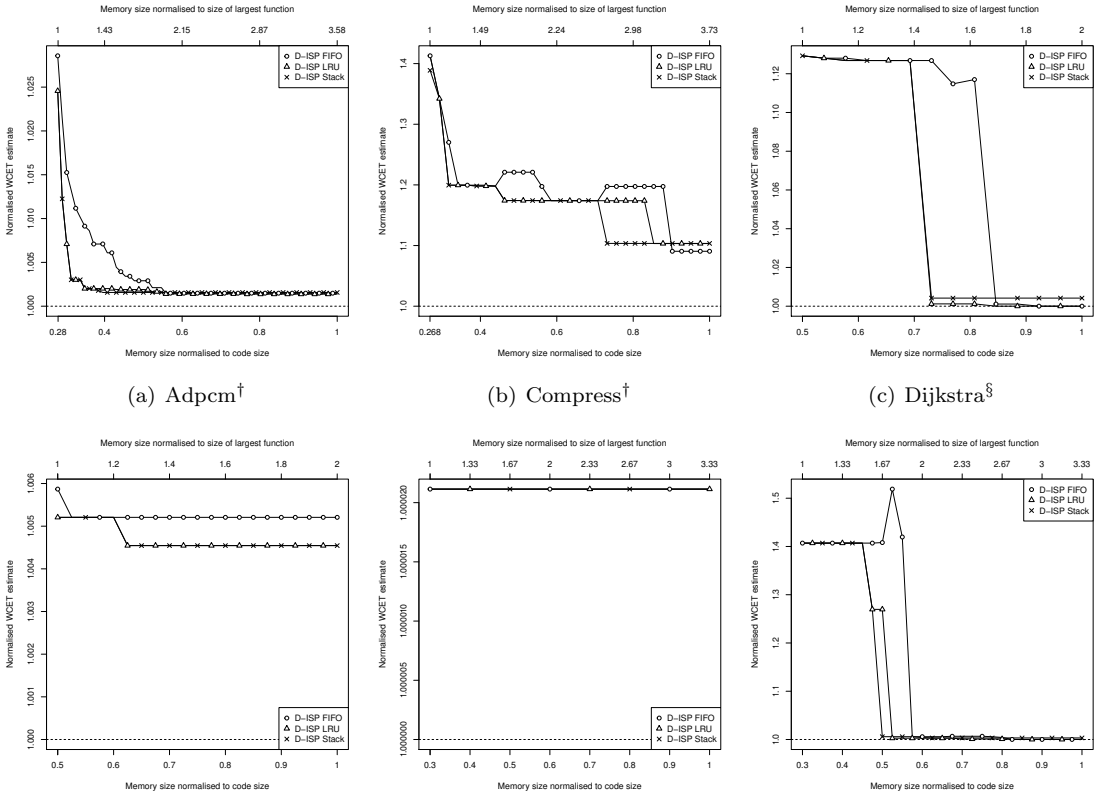
size is increased in steps of 32 B until the size of the benchmark is reached. The memory sizes are normalised to the benchmark's code size. In the Figures 6.20(a)-(f) the D–ISP is compared to the full associative LRU cache. The solid line shows the WCET estimates of the D–ISP. The dashed line represents the WCET estimates of the LRU cache in a system with shared off-chip memory connection and considering the memory interferences (denoted as SHMC). The dotted (and lower) line shows the estimates for a system with separated off-chip memory connection for which interferences does not need to be considered (denoted as SEMC). The gap between SHMC and SEMC spans the range in which the real WCET of a system with shared off-chip memory connection will be located, because the SHMC value overestimates the WCET, by assuming every off-chip memory access interferes, and the SEMC value will underestimate the WCET, since interferences will occur. Because the D–ISP eliminates the interferences at the off-chip memory level by design, the WCET estimates of the D–ISP are independent of the used off-chip memory connection in the system, i.e. a SHMC and SEMC system have the same WCET estimate when using the D–ISP. So, all WCET estimates of the D–ISP in the area between the SHMC and SEMC cache estimates deliver tighter WCET estimates than a cache in a shared off-chip memory hierarchy (SHMC), due to the absence of interferences. But it is unknown if the real WCET can be undermatched by the D–ISP. If the estimates are below the SEMC estimate for a cache, the D–ISP clearly outperforms a system with shared off-chip memory connection *and* a system with separated off-chip memory connection. For estimates above SHMC, the D–ISP cannot deliver lower WCET guarantees than a cache in any configuration of the off-chip memory connection.

In summary it could be shown that the D–ISP is able to exploit the absence of memory interferences to provide lower estimates than a system with cache and shared off-chip memory connection, if the interference penalty is charged for every off-chip memory access. An integrated analysis that estimates at which points in the program different memory accesses actually interfere is able to reduce the WCET estimates for such system. In this case the WCET estimate for a SHMC system approximates SEMC estimate, but is not able to reach it, because memory interferences will occur in a shared off-chip memory hierarchy. So any WCET estimate of the D–ISP that is slightly above the SEMC estimate is unlikely to be undermatched by a precise integrated analysis for a system with shared off-chip memory connection. Hence, the D–ISP is able to provide lower WCET estimates with a less complex analysis than an integrated WCET analysis for a cache in a system with shared off-chip memory connection. Using the stack-based replacement policy instead of FIFO the D–ISP can reduce the WCET estimates even further as the Tables B.2 to B.16 in Appendix B.2 and the discussion in the following section shows.

6.2.6 Comparison of Different D–ISP Replacement Policies

In Section 6.2.3 the WCET estimates of the D–ISP with FIFO replacement policy are compared to the estimates of caches and static scratchpads. In this section the different replacement policies of the D–ISP that are described in Section 3.2.4 (FIFO, LRU, and stack-based), are compared with each other. Therefore, the ISPTAP tool performs the analysis for the LRU and stack-based replacement policies as described in Section 4.3. The evaluation methodology and the used benchmarks are the same as for the FIFO D–ISP in Section 6.2.3 that is described in Section 6.2.1. The only difference is how the content in the scratchpad is managed. The timing of the CarCore processor and the D–ISP controller and also the memory access times (MMAT) are not changed. Thus the WCET estimates of the D–ISP with FIFO replacement policy shown in that Section 4.3 (and in Appendix B) are the same as in the WCET estimates presented here.

The results of the comparison of the different D–ISP replacement policies are shown in the Figures 6.21 and 6.23. These figures show the WCET estimates normalised to the WCET of a



(a) Adpcm† (b) Compress† (c) Dijkstra§

(d) Edn† (e) Matmult† (f) Sha§

Figure 6.21: Comparison of the D–ISP replacement policies for Mälardalen/MiBench suites (examined policies: FIFO, LRU, and stack-based)

system in which the code is completely located in on-chip memory. The baseline WCET estimates in cycles is shown in the Table B.1. The WCET estimates for the different replacement policies are calculated for D–ISP sizes starting at the size of the largest function in the benchmarks up to the code size of benchmark in steps of 32 B. In the figures the D–ISP size is normalised to the code size of the benchmarks at the lower x-axis and to the size of the largest function at the upper x-axis (refer to Table 6.12 for the sizes in B).

The Figure 6.21 shows the normalised WCET estimates of the Mälardalen (denoted with †) and the MiBench (denoted with §) benchmark suites. The estimates for the EEMBC benchmarks (denoted with ‡) are shown in Figure 6.23. Notice that the *Loop* configurations of the benchmarks are not shown, because they behave like the unmodified benchmarks. In general it is shown for all benchmarks that the LRU and the stack-based replacement policies outperform the FIFO policy. This holds especially for smaller D–ISP sizes, because the caller function will be less likely evicted in LRU (because it is recently accessed) and in the stack-based replacement policy (because it is kept on cost of other callee functions) than in FIFO. By overwriting a callee function instead of using possibly unused memory space the stack-based replacement policy might not reach the optimal performance in contrast to LRU and FIFO.

The Figure 6.21(a) shows for Adpcm that lower WCET estimates are reachable, if a large caller function will not get evicted. In Adpcm two larger functions call multiple smaller functions one after the other. So FIFO evicts the older and larger caller function frequently to keep the lately loaded functions, but since the execution always returns to the large caller functions this is not optimal. Therewith, FIFO gets outperformed by LRU and the stack-based replacement policy. Both replacement policies have a similar performance and reach close to optimal WCET estimates with smaller scratchpad sizes than FIFO.

For Compress depicted in Figure 6.21(b) the WCET estimates of the FIFO D–ISP increase at two points when the scratchpad size is increased. This is, as discussed before, due to unclassified memory states. For LRU and stack-based this effect does not occur, because recently activated functions are "moved" to the front of the scratchpad as for LRU or the replacement policy is more restrictive as for stack-based. Hence, when using a D–ISP with LRU or stack-based replacement policy, the WCET estimate monotonically decreases with increasing scratchpad size. In Figure 6.21(b) it is also shown that the stack-based policy is able to outperform LRU, because LRU evicts a caller function that is not as as frequently used as another function, but more costly to load. Furthermore, it is shown that the stack-based replacement policy cannot reach the same estimates as FIFO for large scratchpads, because on call another callee gets evicted, even if enough free memory is available. But this is in the philosophy of the stack-based replacement policy that rather keeps a caller than a callee (refer to Section 3.2.4). The reason for the lower WCET estimates for FIFO than for LRU for larger memory sizes is the additional impreciseness induced by the usage of must and may set for the LRU analysis, whereas the analysis of the FIFO policy maintains all concrete memory states[27].

Also for the Dijkstra benchmark the stack-based and the LRU replacement policy are able to reach the their optimal WCET estimates with smaller memory sizes than FIFO. Figure 6.21(c) again shows that the stack-based replacement policy is not as optimal for relatively large scratchpad sizes and it is outperformed by FIFO and LRU for sizes larger than 704 B. Anyhow, it reaches its optimal WCET estimate at the same scratchpad size as LRU (which is 608 B).

Edn and Matmult shown in the Figures 6.21(d) and 6.21(e) respectively have a very similar[28] or even the same WCET estimates for all replacement policies. This is caused to the structure of these applications and the D–ISP's capability to load small functions in the slipstream while the call/return microcode is processed (refer for details to Section 6.2.3). So no additional penalty is charged, if a small function is reloaded.

The comparison of the three different D–ISP replacement policies for the Sha benchmark (Figure 6.21(f)) shows lower WCET estimates for the LRU and stack-based replacement policy for small scratchpad sizes than the estimates for FIFO. The benchmark code contains a WCET dominating loop in which the function `sha_transform` calls the function `FUNC` frequently, which again calls one of four functions (`f1` to `f4`) and twice the rotation function (`ROT32`). The call graph of Sha is shown in Figure 6.22. The functions called by `FUNC` are rather small, whereas the other two dominating functions are significantly larger. For the FIFO replacement policy the large functions get evicted on loading the small ones, until all 7 functions fit in the D–ISP (at scratchpad size of 736 B). Since the larger functions `FUNC` and `sha_transform` are frequently activated (due to the loop in `sha_transform`), these functions should be kept in the D–ISP memory. Unfortunately, the function `ROT32` will also kept in the memory by the LRU policy, because is activated twice on each call of `FUNC`. So the close to optimal WCET estimate for the LRU D–ISP is reached at the scratchpad size of 672 B such that the two major functions, the

[27] One call site of the function `cl_hash` cannot be classified as always hit for LRU that is a sure hit in the FIFO memory analysis. Thus the function load penalty is charged for LRU and not for FIFO.

[28] For Edn the difference in the WCET estimates between the replacement policies is about 60 cycles, which is less than 0.1% of the overall WCET.

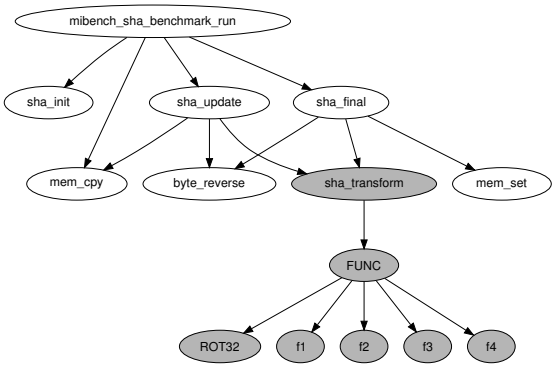
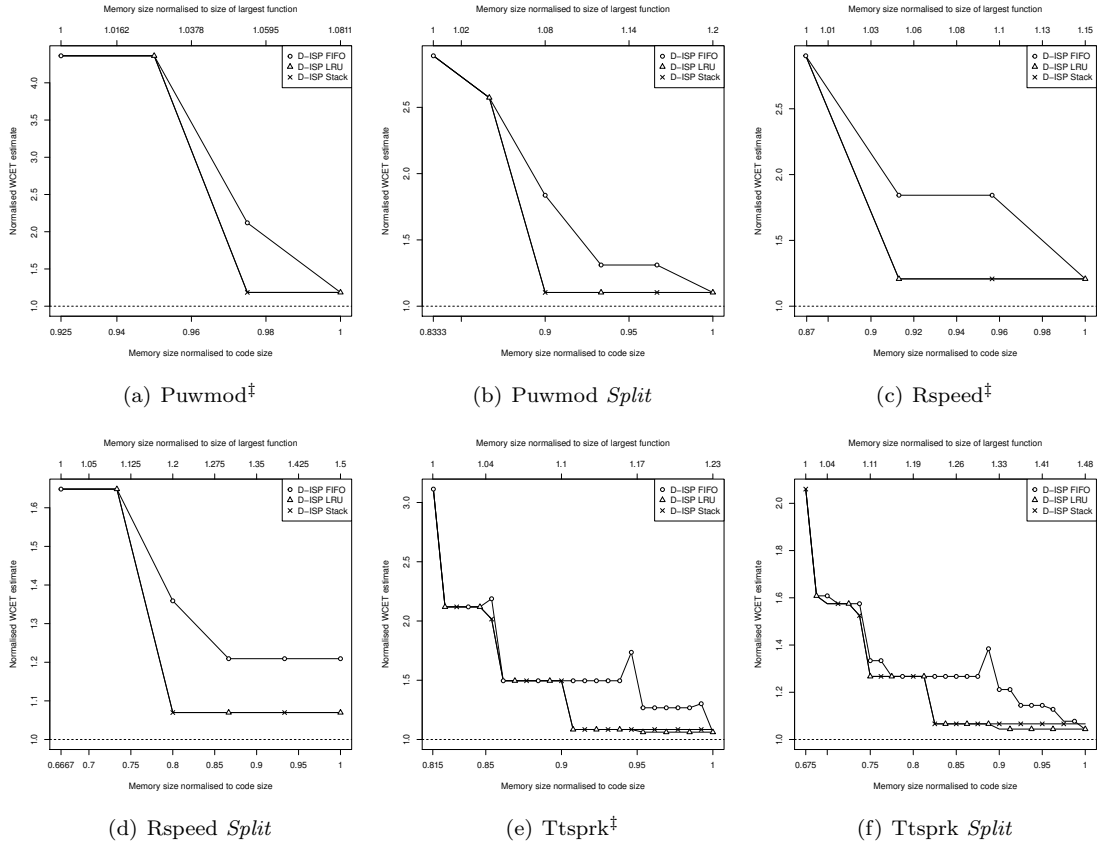


Figure 6.22: Call graph of Sha with highlighted WCET dominating functions

rotation function and the largest function of `f1` to `f4` (which is `f3`) fit the scratchpad. The stack-based replacement policy always evicts on function call any previously called function at the same call level. Therefore, on execution of the function `FUNC` for example the function `ROT32` is evicted on call of function `f1` and `f1` itself will be evicted when `ROT32` is called again later. For Sha this results in the fact that the two large functions are kept in the D–ISP, whereas only the small child functions of `FUNC` get evicted during execution. Hence, the stack-based replacement policy reaches its optimal estimate already at the D–ISP size of 640 B, which is suitable to maintain the functions: `sha_transform`, `FUNC`, and the largest callee function of `FUNC` (which is again `f3`). The stack-based replacement policy leads to no further improvement of the WCET estimate when continuing to increase the D–ISP size. This results in a non-minimal WCET estimate compared to FIFO or LRU for large scratchpad sizes.

For the EEMBC benchmarks presented in Figure 6.23 the FIFO replacement policy of the D–ISP gets also outperformed by LRU and the stack-based replacement policy. The Figures 6.23(a) to (d) illustrate that for Puwmod and Rspeed. Further for both benchmarks the WCET estimates for LRU and the stack-based replacement policy are equal. Examining Ttsprk (Figure 6.23(e) and (f)) shows the estimates differ only for larger D–ISP sizes in which the stack-based replacement policy fails to utilise the given scratchpad size.

To summarise the comparison of the three different D–ISP replacement policies, the WCET estimates for FIFO are not as good as for the other policies, but FIFO can be implemented with the lowest hardware complexity. The LRU and the stack-based policy deliver similar WCET estimates especially for smaller scratchpad sizes. For larger scratchpad sizes the stack-based policy can get outperformed due to the eviction of functions that are at the same call level as the called function. But in contrast to LRU, the stack-based replacement policy can be implemented in hardware (refer to Section 3.3.4) with an overhead of up to 25% comparing to the FIFO replacement policy. The hardware overhead of the stack-based replacement policy was discussed in detail in Section 6.1.5. So for smaller scratchpad memory sizes the stack-based replacement policy is favourable (even with the additional hardware complexity it requires), whereas FIFO is the replacement policy of choice for large scratchpad memory sizes. The LRU replacement policy is known for the superior average case performance for common caches [Al-Zoubi et al., 2004; Hennessy and Patterson, 2006] and it also is the suggested replacement policy for caches in hard real-time systems [Reineke et al., 2007]. However, it is shown that for a dynamic function-based memory like the D–ISP LRU cannot clearly outperform a stack-based replacement policy. Though there are possible scenarios in which LRU can deliver lower WCET estimates, e.g. if a function calls disjoint deep call trees in a nested loop, but on the



(a) Puwmod‡ (b) Puwmod *Split* (c) Rspeed‡

(d) Rspeed *Split* (e) Ttsprk‡ (f) Ttsprk *Split*

Figure 6.23: Comparison of the D–ISP replacement policies for EEMBC benchmark suite (examined policies: FIFO, LRU, and stack-based)

other hand e.g. for Compress and Sha the stack-based replacement policy beats the LRU policy. Therefore, the self-evident assumption that a stack-based replacement policy is the adequate replacement policy for a dynamic function-based memory like the D–ISP is verified for the set of chosen benchmarks and in terms of the estimated WCET. Furthermore, Preußer et al. [2007] shows that this assumption also holds for the average case performance for a function-based cache.

6.2.7 Impact of the Function Lookup Width on the WCET Estimates

In the previous estimations of the WCET for a system with D–ISP it is considered that the function lookup in the *content management* of the D–ISP always takes one cycle. That means that the function lookup width (no_{look}) is as large as the size of the mapping and lookup tables (no_{func}). But as stated in Section 6.1.2 (Table 6.4, p. 162) the hardware cost of the D–ISP controller directly depends on the function lookup width, which specifies the number of functions that can be checked within one cycle during hit/miss detection. To allow a rating of the hardware complexity of the D–ISP controller regarding the estimated WCET in the following the WCET impact of smaller lookup widths is examined.

Table 6.17: Multi-cycle lookup delay of the function activation time on hit/miss with different function lookup widths (according to Table 6.10)

Function Lookup Width no_{look}	**2**	**4**	**8**	**16**	**32**
Delay of function activation due to multi-cycle lookup (in Cycles)	15	7	3	1	0

The baseline for the evaluation is the D–ISP with FIFO replacement policy, a mapping/lookup table size (no_{func}) of 32 functions, and a lookup width (no_{look}) of also 32 functions. Thus the D–ISP controller is able to determine, if an activated function is a hit or a miss within one cycle. For smaller lookup widths multiple cycles are needed to detect a hit or a miss on function activation (so called multi-cycle lookup), if the mapping/lookup table size is not reduced. So for the different lookup widths (from 2 to 32) examined in this section the mapping/lookup table size stays constantly at 32 functions. This table size is suitable, because every benchmark that is used contains less than 32 functions (see Table 6.12).

The multi-cycle lookup is discussed in detail in Section 3.2.3. The implementation of the multi-cycle lookup in the D–ISP controller is conceptually shown in the Section 3.3.4. The Equations (3.10) and (3.11) calculate the activation time of a function in case of hit and miss depending on the lookup width (no_{look}) and the mapping/lookup table size (no_{func}). The Table 6.17 gives an overview on the additional cycles needed for the hit/miss detection, if the lookup width is reduced. The baseline is again the D–ISP with a lookup width of 32 for which the latencies are presented in Table 6.10.

Using these additional delays for every function activation the WCET impact of smaller lookup widths (and thus less hardware complexity) can be quantified by ISPTAP. Therefore, the WCET estimates for the benchmarks introduced in Table 6.12 are examined with different lookup widths. Notice that except from the D–ISP hit/miss detection time every other parameter is left unchanged and thus is same as in the description of the architectural configuration from Section 6.2.1. The WCETs are estimated for each benchmark for the D–ISP sizes starting at the size of the largest function up to the size of the whole benchmark in steps of 32 B. Hence, the results for the lookup width of 32 are the very same as presented in Section 6.2.3. The WCET overhead due to smaller lookup widths shown in Table 6.18 is obtained by normalising the estimates to the lookup width of 32 (single-cycle lookup) and calculating the average mean of the normalised estimates of all D–ISP memory sizes per benchmark.

It can be seen in the Table 6.18 that the impact on the WCET estimate caused by smaller lookup widths is low, below 1% for most benchmarks and a lookup width of 4. This is because the function lookup has to be done only on function activation. During function execution and function load, which are the dominating parts in the WCET estimate, the function lookup latency is irrelevant. So slower function lookups are suitable for the D–ISP and do not add a significant penalty to the overall WCET estimate. By taking the hardware complexity of the D–ISP controller into account (refer to Table 6.4) the optimal trade-off between the estimated WCET and hardware complexity is a lookup width of 4 functions and a mapping/lookup table size of 32 functions for the considered set of benchmarks. A smaller lookup width noticably increases the WCET estimates of the D–ISP and safes only a few ALUTs and registers. Larger lookup widths pose a significantly higher hardware complexity in terms of ALUT and register count and the reduction of the WCET estimates is only marginal. As a result of this evaluation the lookup width of 4 functions is to be considered as best in terms of estimated WCET and required hardware cost.

Table 6.18: Average impact of the lookup width on the WCET estimate (For lookup widths (no_{look}) from 2 to 16 and a mapping/lookup table size (no_{func}) of 32. The baseline is a D–ISP with a lookup width of 32.)

Benchmark	**Function Lookup Width no_{look}**			
	2	**4**	**8**	**16**
Adpcm†	0.03%	**0.01%**	0.01%	≈0%
Compress†	2.50%	**1.11%**	0.47%	0.16%
Dijkstra§	1.53%	**0.71%**	0.30%	0.10%
Edn†	0.11%	**0.05%**	0.02%	0.01%
Edn *Loop*	0.10%	**0.04%**	0.02%	0.01%
Matmult†	0.13%	**≈0%**	≈0%	≈0%
Matmult *Loop*	0.11%	**≈0%**	≈0%	≈0%
Puwmod‡	1.21%	**0.57%**	0.24%	0.08%
Puwmod *Split*	1.87%	**0.80%**	0.33%	0.09%
Rspeed‡	1.94%	**0.90%**	0.39%	0.13%
Rspeed *Split*	4.48%	**1.92%**	0.76%	0.18%
Sha§	1.26%	**0.59%**	0.25%	0.08%
Ttsprk‡	0.58%	**0.27%**	0.12%	0.04%
Ttsprk *Loop*	0.56%	**0.26%**	0.11%	0.04%
Ttsprk *Split*	0.75%	**0.34%**	0.15%	0.05%

6.3 Impact on the Average Case Performance

Beside the WCET performance aspect of the D–ISP in this section the performance for the average case is discussed. The average case performance is worth a deeper look to classify how the D–ISP behaves in an embedded system during normal operation, in which the WCET-critical path is not necessarily traversed. Therefore, the fetch cost of several benchmarks is obtained by a cycle-accurate simulation of a system with D–ISP and compared with systems that employ other commonly used instruction memories. The impact of the D–ISP on the average case performance is also published in [Metzlaff et al., 2011a]. For a deeper look into the discussion of the average case performance for scratchpads and caches the reader is referred to [Angiolini et al., 2004; Janapsatya et al., 2006b] and to [Milenkovic et al., 2003; Al-Zoubi et al., 2004], respectively.

6.3.1 Evaluation Methodology

As stated in Section 3.3.2 the D–ISP was implemented in a cycle-accurate SystemC simulator for architectural evaluation. So for the determination of the average performance the SystemC model of the CarCore processor including the D–ISP is used. The executed code is instrumented with the FLE instructions (see Section 5.1) such that the D–ISP is aware of the sizes of the functions in the code.

The D–ISP performance is compared to the performance of an instruction cache and a static instruction scratchpad. The instruction cache is direct mapped and has a cache line size of 32 B (according to the instruction cache specifications of the TriCore processor [TriCore, 2004]) containing 4 64 bit fetch blocks each. The static instruction scratchpad contains multiple functions of the benchmark code. For the selection of the functions that are put into the scratchpad the knapsack optimisation to is used to fill the given scratchpad size as best as possible. As metric for selection of the content the estimated WCET is chosen. Refer to Section 4.1.2 for the descrip-

tion of the used algorithm. This static scratchpad is denoted in the following as FS–ISP (KN). Functions that are not contained in the FS–ISP are fetched directly from the off-chip memory.

The memory access time for the FS–ISP is one cycle. For fetches that lead to the off-chip memory level 4 cycles are needed. As a scratchpad access, a cache hit also takes one cycle. On a cache miss four 64 bit off-chip memory accesses are needed plus one extra cycle to detect the miss. For D–ISP the hit detection on call or return cost 4 cycles for table lookup and context register write. This delay is hidden by the call or return processing of the pipeline. A miss of the D–ISP takes as much cycles as are needed to load the complete function from the off-chip memory plus 5 cycles for internal processing, like table lookup, context register write, and write latencies. Fetches are handled by the D–ISP within one cycle. This latencies correspond to the values used to calculate the MMAT in Table 6.9 of Section 6.2.1. For the D–ISP the same parameters as shown in Table 6.10 are used. Since the average case performance analysis is not as important as the impact of the D–ISP on the WCET, in the following evaluation only the D–ISP with FIFO replacement policy is discussed.

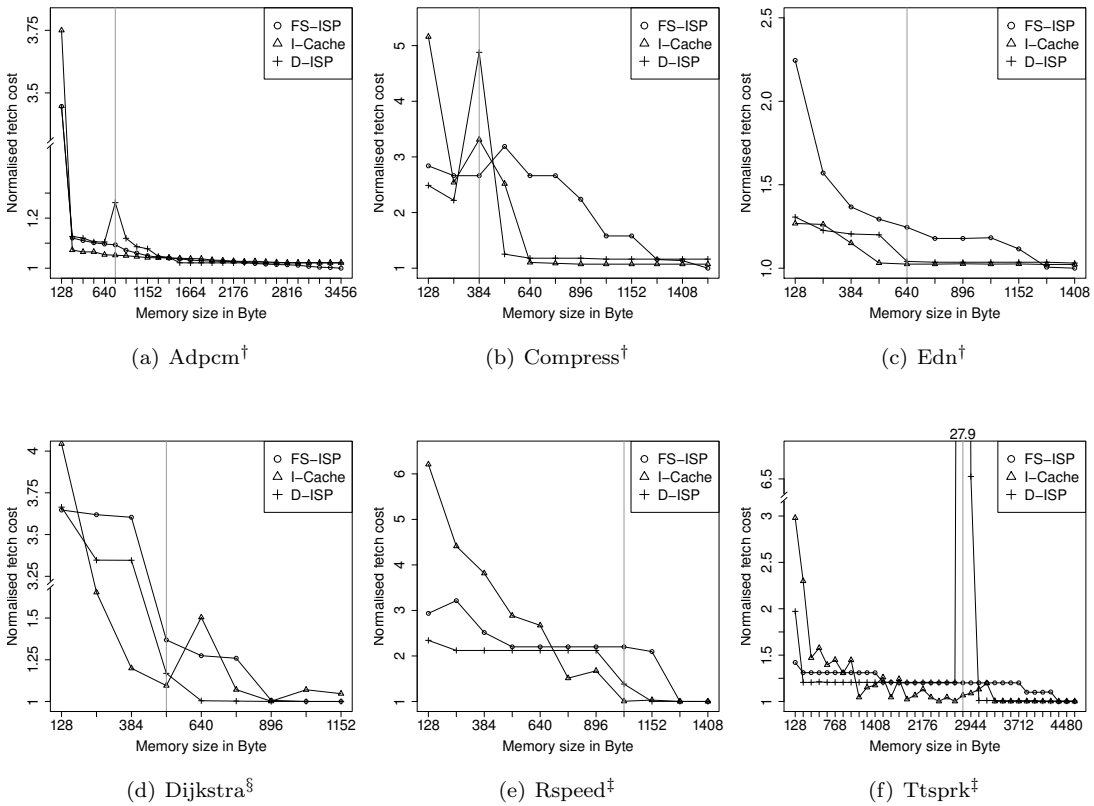
For the performance evaluation the fetch cost of applications from three different benchmark suites are measured: Mälardalen WCET benchmark suite [Gustafsson et al., 2010] (marked with †), EEMBC AutoBench 1.1 [Poovey et al., 2009] (marked with ‡) and the MiBench suite [Guthaus et al., 2001] (marked with §). A selection 6 benchmarks with different function length and call characteristics is chosen from the three suites: Adpcm†, Compress†, Edn†, Dijkstra§, Rspeed‡, and Ttsprk‡. The general characteristics of these benchmarks are already described by Table 6.12 of Section 6.2.1.

The benchmarks used for the average performance evaluation have a larger code size than the ones used for WCET impact in Section 6.2. This is because beside the benchmark function also the main entry point of the program (function `main`) and the whole initialisation code needs to be simulated. Another difference of the used benchmarks in the two evaluations affects only the EEMBC benchmarks: The version that is analysed to obtain the WCET estimate (refer to Section 6.2.1) moved the parameter initialisation into an extra function that sets the values needed for the benchmark algorithm into a harness structure. In the original code the initialisation is embedded in the benchmark function and uses normal variables for the parameters (located on the stack). The relocation of the initialisation code into another function is done to obtain the WCET estimate of the real benchmark code only, without considering the initialisation code. This change increases the size of largest function, because the accesses of the parameters in the harness structure need additional address calculation. Since the evaluation of the average case performance presented here uses the original benchmark code, the sizes of the largest functions are slightly smaller compared to the evaluation of the WCET impact. Anyway, the application behaviour is the same for both benchmark versions.

The fetch cost measured by the SystemC simulator represents the number of cycles required for all fetches requested by the processor for the whole benchmark run. The fetch cost in cycles is normalised to the configuration where the complete code is located in the FS–ISP, which has the minimal possible fetch cost (normalised fetch cost of 1). A normalised fetch cost of 4 will be reached, if all instructions are fetched from the off-chip memory. To compare the three on-chip instruction memories their sizes are varied from 128 B to the overall size of the benchmark in 128 B steps.

6.3.2 Results

The normalised fetch cost for the execution of the different benchmarks is shown in Figure 6.24. It depicts that, if the memory size is as large as the benchmark code, the FS–ISP always performs slightly better than the cache and the D–ISP. This is caused by the fact that the content of



(a) Adpcm† (b) Compress† (c) Edn†

(d) Dijkstra§ (e) Rspeed‡ (f) Ttsprk‡

Figure 6.24: Normalised fetch cost for FS–ISP (KN), direct mapped cache, and D–ISP with different memory sizes

FS–ISP is set up before the benchmark's execution, which was already discussed in Section 6.2.4 for the WCET estimation. The fetch cost of the FS–ISP is reduced with increasing scratchpad sizes, because more functions can be assigned to the scratchpad. But there are exceptions for this behaviour in Compress (b) and Rspeed (e). They are caused by the used metric of the static assignment algorithm that takes the longest possible path for each function and the maximum number of function invocations into account, though this might not reflect the real execution behaviour.

The cache reduces the fetch cost to nearly the optimum even with a memory size lower than one third of the code size. But in some configurations the cache shows a thrashing behaviour, like for Compress (b) at 384 B. This is caused by its direct mapped organisation. Also the cache miss rate for the EEMBC benchmarks (e) and (f) is high. This is due to the low spatial locality of the EEMBC benchmark caused by code replication.

In the Figures 6.24(a) to (f) the upright lines denote that the memory size is at least as large as the size of the largest function in the benchmark code. This mark is important for the D–ISP, because it loads only complete functions into the scratchpad. Thus, for any measures on the left-hand side of the upright line, the D–ISP has to ignore functions larger than its size. This is done automatically by the D–ISP controller. For configurations with a larger (or equal) scratchpad sizes than the largest function every fetch request is directed to and handled by the

D–ISP. It is shown that the D–ISP also reaches a reasonable average case performance, if the scratchpad size is below the size of the largest function. But then the timing analysis cannot assume the absence of memory access interferences between data and instruction memories, because the D–ISP can only enforce the two-phased execution scheme, if it maintains the active function. Therefore, from hard real-time point of view a D–ISP that is smaller than the largest function in the code should not be considered.

For benchmarks (a), (b), and (f) some outliers for the D–ISP appear, if the size of the scratchpad is slightly larger than the largest function. This is caused by the fact that in those cases the largest function will always evict nearly all other functions maintained in the scratchpad on its load. So if the largest function is very active in calling or getting called, the fetch cost for the D–ISP will increase, caused by evicting and reloading functions frequently. For Ttsprk (f) this behaviour results in a normalised fetch cost of 27.9 which is almost 7 times worse than using no on-chip memory[29]. The bad performance of the D–ISP is caused by the flat call hierarchy of Ttsprk and the fact that every function is called directly by the large main function.

For scratchpad sizes that are not close to the size of the largest function the D–ISP performs better than the FS–ISP. This is explainable by the dynamic content managed of the D–ISP, which allows the D–ISP to maintain the functions that are used in the actual phase of the application, whereas the FS–ISP contains a fixed set of functions during the whole application run. Comparing the D–ISP to the instruction cache, the cache accounts mostly a lower fetch cost. This is caused by the finer granularity of the cache lines that hide the structure of the application. Thus, the D–ISP cannot compete to the instruction cache in many configurations, since the cache needs to load only one line on a miss, whereas the D–ISP has to load the complete function. This is a performance drawback of the D–ISP, because on function call or return usually only a fraction of the activated function is executed.

In general the average performance of the D–ISP is between the FS–ISP and cache performance. This is expected since it uses a dynamic memory management that is not as fine-grained as for a cache. Nevertheless, the decent average case performance of the D–ISP is clearly outweighed by the estimated WCET performance that was evaluated in Section 6.2.3.

6.4 Complexity of the D–ISP Memory Analysis

In this section the complexity of the instruction memory cost analysis (refer to Figure 5.4 for the analysis steps of ISPTAP) of the D–ISP in terms of memory usage and analysis time is discussed. Therefore, the state-of-the art LRU cache analysis of Ferdinand and Wilhelm [1999] (refer to the Section 4.2.2) using abstract interpretation is taken as competitor. For the D–ISP with FIFO replacement policy ISPTAP has to maintain all possible concrete memory states during content analysis, because no safe abstraction of the memory states could be found (see Section 4.3.2). For the evaluation shown in this section WCET analyses of different benchmarks with D–ISP and cache as instruction memory were performed with ISPTAP.

6.4.1 Analysis Memory Cost

The Table 6.19 shows the complexity of the D–ISP and cache analysis in terms of maintained memory states and size of the used memory for the set of benchmarks described in Table 6.12. In the second column the number of states needed by ISPTAP for the analysis of the D–ISP with FIFO replacement policy is presented. In column three the memory amount is provided

[29]Then all memory accesses are handled by the off-chip memory with a MAT of 4 cycles, which results in a normalised fetch cost of 4.

Table 6.19: Comparison of the analysis complexity for FIFO D–ISP and LRU cache (The FIFO D–ISP analysis uses all concrete states, whereas the cache analysis keeps only the abstract *must* and *may* set.)

Benchmark	**D–ISP$_{FIFO}$**		**I–Cache$_{LRU}$/D–ISP$_{FIFO}$**	
	Avg. No. of States	**Avg. Used Memory**	**Avg. No. of States**	**Avg. Used Memory**
Adpcm	170	19.8 KiB	7.82	20.51
Compress	236	18.8 KiB	2.37	5.14
Dijkstra	177	15.6 KiB	4.05	6.10
Edn	20	1.9 KiB	12.30	16.63
Edn *Loop*	44	4.4 KiB	11.82	19.19
Matmult	28	2.5 KiB	9.43	7.03
Matmult *Loop*	60	5.5 KiB	9.27	7.36
Puwmod	160	12.4 KiB	3.71	6.59
Puwmod *Split*	154	11.9 KiB	3.04	5.42
Rspeed	32	2.7 KiB	5.63	6.64
Rspeed *Split*	41	3.5 KiB	6.44	6.57
Sha	37,084	3,399.7 KiB	0.11	0.20
Ttsprk	603	47.4 KiB	3.09	14.02
Ttsprk *Loop*	1,729	134.0 KiB	2.17	12.54
Ttsprk *Split*	661	52.8 KiB	2.97	11.48
Average	2,747	248.8 KiB	5.61	9.69

that ISPTAP needs to store these D–ISP memory states during analysis (later also denoted as memory footprint of the analysis). The values shown are obtained by analysing the different benchmarks with D–ISP sizes from the size of the largest function to the benchmarks code size in steps of 32 B. In the table the average mean of the measures from all configurations is presented. To allow a judgement of the D–ISP analysis complexity these measures are set into relation with measures of the LRU cache analysis for the same benchmark. For the LRU cache analysis the average mean of the number of abstract memory states and the amount of memory to maintain them for all cache sizes ranging from 32 B (one cache line) up to the size of the code in steps of 32 B is calculated per benchmark. Then these measures of the cache and the D–ISP are divided to show the state and memory overhead of the cache (see columns 4 and 5 of the Table 6.19).

As shown in the table the amount of D–ISP memory states and the memory footprint of the analysis is limited: except for Sha the number of D–ISP states is below 2,000 and ISPTAP requires less than 150 KiB to maintain the D–ISP states. The complexity of the cache analysis is for almost all benchmarks higher, because the cache analysis has to hold the abstract memory states for each basic block. In contrast to that the D–ISP analysis needs to hold the D–ISP content on function call and return only. Thus, for the cache analysis the number of memory states and the necessary amount of memory is higher than for the D–ISP analysis. The additional need of the cache analysis ranges from 2 to 12 times more memory states than for the D–ISP analysis. The increased number of memory states also lead to a bigger memory footprint. The memory required by ISPTAP to hold all abstract memory states is between 5 to 20 times larger than for the D–ISP.

One exception is the Sha benchmark for which the number of maintained concrete memory states for the D–ISP analysis is about 10 times higher than for the cache analysis. This is caused by multiple nested loops for which in each iteration depending on the processed data a different

Table 6.20: Comparison of the analysis complexity for FIFO D–ISP and LRU D–ISP (The FIFO D–ISP analysis uses all concrete states, whereas the LRU D–ISP analysis keeps only the abstract *must* and *may* set, as proposed in Section 4.3.1.)

Benchmark	**D–ISP$_{LRU}$/D–ISP$_{FIFO}$**	
	Avg. No. of States	**Avg. Used Memory**
Adpcm	2.00	1.74
Compress	0.70	0.50
Dijkstra	1.24	0.86
Edn	2.00	1.35
Edn *Loop*	1.95	1.43
Matmult	2.00	1.15
Matmult *Loop*	1.97	1.20
Puwmod	1.34	0.69
Puwmod *Split*	1.12	0.64
Rspeed	2.00	1.00
Rspeed *Split*	2.00	1.01
Sha	0.031	0.025
Ttsprk	1.13	0.64
Ttsprk *Loop*	0.79	0.49
Ttsprk *Split*	1.05	0.62
Average	1.42	0.89

function can be called (refer also to the call graph in Figure 6.22). By virtually unrolling all loops the state space that the D–ISP analysis needs to handle in this benchmark explodes. Despite that, the analysis is done for all D–ISP sizes within less than 20 seconds on an AMD Opteron 6100 series.

Due to the explosion of memory states and the larger memory footprint for a more complex benchmark, it is very likely that the analysis of the FIFO D–ISP that maintains all concrete states does not scale for real-world applications. To cope with the analysis complexity for larger applications an analysis that reduces the number of maintained states is required. For caches the reduction to an abstract *must* and *may* set is a common method to reduce analysis complexity. Unfortunately, a sound analysis for the FIFO D–ISP using abstract interpretation could not be found within this work, but it was possible to adapt the LRU cache analysis for a D–ISP with LRU replacement policy (shown in Section 4.3.1).

The results for the complexity of the LRU D–ISP analysis are shown in Table 6.20. The number of memory states maintained by the analysis and the size of the used memory are normalised to the values of the FIFO D–ISP. For benchmarks as Adpcm or Edn for which the FIFO D–ISP analysis has to maintain only a small number of concrete states the use of the abstract LRU D–ISP analysis increases the state count and also the required memory. This is because the LRU D–ISP analysis always maintains a *must* and a *may* set, containing all functions that have to or might be in the D–ISP, respectively. Hence, the number of states is increased at most by the factor of 2, but this can only happen for very simple benchmarks. The complexity of the FIFO analysis can only compete with the LRU analysis, if each function call or return can be reached by at most two paths with different D–ISP memory content. But if in the application many different paths (with different D–ISP content) are possible, the complexity of an analysis that maintains all possible concrete D–ISP states increases drastically. In those cases an analysis

that maintains only two abstract memory states, namely *must* and *may*, can significantly reduce the amount of necessary memory states and also the memory footprint. For example the amount of memory states and the memory footprint needed by the LRU D–ISP analysis could be reduced for Sha to approximately 3% of the original values required by the FIFO D–ISP analysis.

So with the LRU D–ISP analysis a more scalable analysis than the FIFO D–ISP analysis that requires to maintain all reachable D–ISP states during analysis is proposed. A precise quantification of benefits of this analysis require the examination of a wider range of especially larger benchmark applications. However, the presented preliminary results strongly motivate the development of more scalable analyses for the D–ISP with FIFO and stack-based replacement policy.

6.4.2 Analysis Time

The analysis time of the D–ISP and the cache with LRU and direct mapped replacement is in general not mentionable, because all different memory sizes of one benchmarks are analysed within at most two minutes on an AMD Opteron 6100 series. Only for the FIFO cache the analysis of three benchmark configurations, which are Sha, Ttsprk *Loop*, and Ttsprk *Split*, is too memory intense to terminate within one hour. Thus, these configurations are highlighted in Table 6.13 (p. 177). The FIFO cache analysis of Sha that tracks all possible concrete states (refer to Section 4.2.3) requires more than 5 GiB of memory and does not even terminate within two weeks. Hence, no WCET estimates could be provided for the FIFO cache in Figure B.12(b).

For the S–ISP memories the most time consuming part of the WCET analysis is the finding of the scratchpad assignment using the ILP formulations given by Section 4.1. In general the S–ISP assignment is found within less than half a minute per S–ISP size. But there are configuration in which the ILP solving takes significantly longer, especially for the complex ILP formulations the KNJ and WSJ algorithms generate for the BBS–ISP. Anyhow, the analysis of all S–ISP sizes per benchmark takes more than one hour for only four benchmark/algorithm configurations: Adpcm BBS–ISP with KNJ, Ttsprk BBS–ISP with KNJ and WSJ, and Ttsprk *Loop* with KNJ. These configurations are also highlighted in Table 6.13.

6.4.3 Conclusion

The D–ISP analysis applied for the FIFO and the stack-based replacement policy, which maintains all reachable concrete memory states during the analysis, is suitable for the set of chosen benchmarks. But as the development of the analysis complexity for the Sha benchmark shows, such analysis is not capable to handle larger applications efficiently. The fact that the D–ISP analysis has to track the memory states only on call and return and not for each basic block as in a cache analysis mitigates the complexity of the analysis and thus assures its appliance for rather small benchmarks. For larger benchmarks, as e.g. real-world applications, a more scalable D–ISP analysis has to be found. The LRU D–ISP analysis that uses only an abstract *must* and *may* set instead of maintaining all reachable memory states points out that an elaborate analysis is able to reduce the analysis complexity of the D–ISP significantly. Unfortunately, the analysis of the FIFO replacement policy is not as easy as for LRU as e.g. Grund and Reineke [2010a] shows. Hence, the development of scalable D–ISP analyses for the FIFO and also the stack-based replacement policy is left for further work.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this work a dynamic function-based instruction scratchpad, the D–ISP, was proposed as first level instruction memory for hard real-time systems. The D–ISP loads functions automatically on demand, such that each function is guaranteed to be in the scratchpad on its execution. Therefore, it uses a dynamic content management that is triggered on function call and return. The D–ISP shows a two-phased behaviour: In the first phase the D–ISP stalls the pipeline and initiates the copying of the activated function into the scratchpad, if it is not already contained. The second phase guarantees that during the execution of a function every instruction fetch will be served by the D–ISP. This scheme results in a predictable and instantaneous execution of a function, once it is loaded.

The D–ISP controller design, consisting of a lightweight process for fetch control and an elaborate content management, was developed and verified using a SystemC simulator. Based on this cycle-accurate simulation model the D–ISP was implemented in an FPGA using the hardware description language VHDL. To allow measuring performance of the D–ISP it was integrated in the CarCore processor and the MERASA multicore processor. Since the estimation of the WCET is crucial to rate the usefulness of micro-architectural features for hard real-time systems, the custom static WCET analysis tool ISPTAP was developed. It contains a timing model of the CarCore processor and a content analysis of the D–ISP. Scratchpads with fixed content and caches, which represent both extremes in terms of predictability and performance, were also examined to compare the proposed D–ISP with common instruction memories used in embedded hard real-time systems. The evaluation of the D–ISP performed in this work is threefold and covers the hardware cost, the impact on the WCET estimates of the system, and a study of the average case performance.

The comparison of the complexity of the D–ISP and common cache implementations in terms of logic amount and memory overhead, performed in Section 6.1, shows that the D–ISP requires a higher hardware amount than associative LRU caches, but the costs are the same order of magnitude. The difference of a cache to the D–ISP is that the D–ISP uses a lookup table, whereas a cache binds each cache line to a cache tag. Therefore, the granularity of the memory used by the D–ISP is decoupled from the amount of memory overhead. Moreover, in contrast to a cache the hit detection in the D–ISP is taken off the instruction fetch path, which results in only a minimal additional delay. Thus, fetches can be handled by the D–ISP with the same memory access time as unmanaged scratchpads.

The hardware effort evaluation of the D–ISP also shows that its complexity strictly depends on the lookup width of the content management, which determines the amount of functions that can be checked within one cycle during hit detection. To mitigate the impact of the hardware amount a multi-cycle lookup was proposed. In Section 6.2.7 it is shown that the slowdown of the function hit detection by the multi-cycle lookup has only a decent impact on the WCET estimates. Hence, the logic amount of a D–ISP with a mapping table size of 32 can reduced to 54% when using a multi-cycle lookup and a lookup width of 4 with only an increase of the WCET estimate of below 2%.

In comparison to other common instruction memories the D–ISP can provide lower WCET estimates. As Section 6.2 showed, the actual improvement depends on the characteristics of the benchmarks. Only if it is possible for the D–ISP to keep the performance critical parts of the application, it is able to reach lower WCET estimates than static scratchpads (BBS–ISP and FS–ISP) and caches with different replacement policies (LRU, FIFO, and direct mapped). On the other hand, if one large function dominates the execution of the whole application, the D–ISP has to provide at least enough memory to host this function. A way to mitigate the impact of one oversized function is to split it into smaller parts, as showed for a set of EEMBC benchmarks. The average improvement of the WCET estimates for the D–ISP in comparison to static scratchpads ranges from -19% for the smallest possible D–ISP size to up to +14% for the maximum memory size. The decent performance of the D–ISP in comparison to the static scratchpads is mainly caused by the fact that the D–ISP has to load the functions during run-time, whereas the scratchpad contains the assigned code already on application start. The impact of the function loading time of D–ISP on the WCET estimates was shown in detail in Section 6.2.4. The comparison of the D–ISP and an LRU cache showed that the D–ISP provides in average 6% to 29% lower WCET estimates for its smallest and largest memory size, respectively. The origin of the improved WCET estimates of the D–ISP is mainly due to the absence of interferences at the shared off-chip memory level, which is unveiled by Section 6.2.5. It is shown that these interferences have a high contribution to the overall WCET estimate, if it is pessimistically assumed that every memory access may interfere. A precise analysis of the interferences at the shared off-chip memory level requires the integration of the instruction and data memory analysis into the pipeline analysis. By the use of the D–ISP the additional complexity that such integration brings is not required to get tight WCET estimates.

Furthermore, in Section 6.2.6 three different replacement policies (FIFO, LRU, and the stack-based policy) were compared for the D–ISP in terms of WCET estimates and hardware effort. The FIFO replacement policy can be implemented with the lowest hardware complexity, but it does not reach the WCET estimates that LRU and the stack-based replacement policy provide, especially for small scratchpad sizes. The WCET estimates for LRU and the stack-based policy are similar, except for larger scratchpad sizes. With larger scratchpad sizes the stack-based replacement policy can get outperformed by LRU, due to the forced eviction of sibling functions. However, it is shown that LRU cannot clearly outperform a stack-based replacement policy. But in contrast to LRU, the stack-based replacement policy can be implemented in hardware. The stack-based replacement policy requires between about 10% and 25% more logic than a FIFO implementation with a lookup width of 32 and 4 functions, respectively. Further, it delivers for small scratchpad sizes noticeable lower WCET estimates than the FIFO replacement policy. For large scratchpad sizes FIFO is able to outperform the stack-based replacement policy. So, the stack-based replacement policy is favourable for small scratchpad memory sizes, whereas FIFO is the replacement policy of choice for large scratchpad memory sizes.

For small benchmarks the complexity of the D–ISP memory analysis performed to estimate the WCET is lower than the cache analysis using abstract interpretation in terms of the analysis memory footprint. This is because the D–ISP content analysis has to maintain the memory states

only on call and return, whereas a cache analysis has to maintain the cache content for each basic block. This results in an in average about 10 times larger memory footprint than the cache analysis using abstract interpretation for the set of examined benchmarks. Unfortunately, the analysis of the D–ISP with FIFO and stack-based replacement policy uses the set of all possible concrete states and is thus not likely to scale for applications with a large amount of functions. However, it was shown in Section 6.4 that this analysis method is suitable for benchmarks of limited complexity. Since the results of the D–ISP analysis showed that this memory is capable of reducing the WCET estimates in comparison to other common memories, the development of a scalable but precise D–ISP content analysis could be beneficial. For the LRU replacement policy an analysis using abstract interpretation was already proposed in Section 4.3.1 and it is shown that the memory footprint of the analysis of more complex benchmarks is reduced significantly.

An average case performance evaluation, presented in Section 6.3, shows that the D–ISP is able to outperform static scratchpad memories, because they cannot profit from a content management and thus have a lower memory utilisation. Due to the fine-grained content management of common caches, the D–ISP cannot compete with their performance. But in general its performance is in the same order of magnitude as the performance of cache memories. Anyhow, the average case performance is much less important in hard real-time systems than the impact on the WCET estimates.

Within this thesis the D–ISP as a promising alternative to common first level instruction memories in embedded hard real-time systems was presented. A set of evaluations were able to proof this claim and outline the characteristics of the D–ISP design and their implications on hardware complexity and WCET estimates.

7.2 Future Work

Since the applicability of the D–ISP in hard real-time systems is shown in this thesis, it would be worth to find a scalable and precise D–ISP analysis. The presented analyses for the D–ISP with FIFO and stack-based replacement policy are not likely to scale for larger real-world applications, because they have to maintain the complete set of all reachable memory states. Approaches based on memory abstractions for the analysis of caches with different replacement policies could be adapted to also support functions with different sizes instead of cache lines of uniform size. Since this generalisation would affect the whole analysis, a sound analysis for the D–ISP is challenging to find.

To allow an optimal D–ISP performance for small and large scratchpad sizes a simple enhancement could be implemented in the D–ISP controller, which allows the support FIFO and the stack-based replacement policy within a single device. This could be done by extending the implementation of the D–ISP with the stack-based replacement policy by a software-controllable register that configures the replacement policy of the content management. Via this register the application can decide if either FIFO or the stack-based replacement policy is used. This is possible, because basically both replacement policies are similar, only on return the replacement policies require a different behaviour. If FIFO is enabled by the software-controllable register, the content management uses the same write and eviction pointers for calls and returns, whereas for the stack-based replacement separated pointers have to be used. So the application designer or a WCET analysis tool could select the most appropriate replacement policy for each application. The evaluation of the hardware effort of such D–ISP controller with hybrid replacement policy is beyond this work, but it is supposed that only minor changes within the content management need to be performed.

The minimal size of the D–ISP memory is specified by the size of the largest function in the code. To avoid this direct dependency of the D–ISP to the application an algorithm could be proposed to split large functions into smaller ones. The main challenge for such algorithm is to find the splitting points with the highest impact on the WCET estimate for the system with D–ISP.

In embedded systems the power consumption is a topic of major importance, because a lower power consumption reduces the thermal dissipation that allows smaller devices with less weight, which saves cost and eases the mounting of the control units in the physical system. On a memory request the D–ISP performs only a simple address mapping instead of a hit detection that common caches require, which is known to be energy intense. Hence, there is the chance that due to this design decision the D–ISP is more energy efficient than a cache. To proof this claim a detailed energy model of the D–ISP has to be build and a comparison to caches is required.

Due to their superior performance and energy efficiency in general purpose systems, multicore processors were also proposed for embedded hard real-time systems, like in [Cullmann et al., 2010; Pitter and Schoeberl, 2010; Ungerer et al., 2010]. It is supposed that the trend of increasing core counts in processors continues, since that in long terms many-core concepts will emerge in hard real-time systems like for example proposed by [d'Ausbourg et al., 2011; Metzlaff et al., 2011b]. Therefore, the implications of the D–ISP in real-time capable many-core architectures are worth to be examined to evaluate if the D–ISP is able to widen the memory bottleneck and allow an increased but still predictable performance of many-core systems.

Bibliography

Aeroflex Gaisler AB. *Leon3/GRLIB SOC IP Library*. http://www.gaisler.com/doc/Leon3%20Grlib%20folder.pdf. [Online, last accessed on 3rd March 2012].

— (Nov. 2010). *GRLIB IP Core User's Manual*. Version 1.1.0.

Akesson, B., K. Goossens and M. Ringhofer (Oct. 2007). 'Predator: a Predictable SDRAM Memory Controller'. In: *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES+ISSS '07)*. New York, NY, USA: ACM, pp. 251–256. DOI: 10.1145/1289816.1289877.

Al-Zoubi, H., A. Milenkovic and M. Milenkovic (Apr. 2004). 'Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite'. In: *Proceedings of the 42nd annual Southeast regional conference (ACM-SE 42)*. New York, NY, USA: ACM, pp. 267–272. DOI: 10.1145/986537.986601.

Alt, M., C. Ferdinand, F. Martin and R. Wilhelm (Sept. 1996). 'Cache behavior prediction by abstract interpretation'. In: *Proceedings of the Third International Symposium on Static Analysis (SAS '96)*. Vol. 1145. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 52–66. DOI: 10.1007/3-540-61739-6_33.

Altera (Aug. 2006). *Stratix II DSP Development Board Reference Manual*. Version 6.0.1.

— (May 2008a). *Cyclone Device Handbook, Volume 1*.

— (Feb. 2008b). *Cyclone II Device Handbook, Volume 1*.

— (July 2009). *Stratix II Device Handbook, Volume 1*. Version SII5V1-4.4.

Altium (Feb. 2009). *TASKING VX-toolset for PCP User Guide*. v3.2. Altium.

Angiolini, F., L. Benini and A. Caprara (Oct. 2003). 'Polynomial-time algorithm for on-chip scratchpad memory partitioning'. In: *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems (CASES '03)*. New York, NY, USA: ACM, pp. 318–326. DOI: 10.1145/951710.951751.

Angiolini, F., F. Menichelli, A. Ferrero, L. Benini and M. Olivieri (Sept. 2004). 'A post-compiler approach to scratchpad mapping of code'. In: *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems (CASES '04)*. New York, NY, USA: ACM, pp. 259–267. DOI: 10.1145/1023833.1023869.

Arnold, R., F. Mueller, D. Whalley and M. Harmon (Dec. 1994). 'Bounding worst-case instruction cache performance'. In: *Proceedings of the 15th IEEE International Real-Time Systems Symposium (RTSS '94)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 172–181. DOI: 10.1109/REAL.1994.342718.

Atanassov, P. and P. Puschner (Dec. 2001). 'Impact of DRAM Refresh on the Execution Time of Real-Time Tasks'. In: *Proceedings of the IEEE International Workshop on Application of Reliable Computing and Communication*, pp. 29–34.

Avissar, O., R. Barua and D. Stewart (Nov. 2002). 'An optimal memory allocation scheme for scratch-pad-based embedded systems'. In: *ACM Transactions on Embedded Computing Systems (TECS)* 1 (1), pp. 6–26. DOI: 10.1145/581888.581891.

Ballabriga, C. and H. Cassé (July 2008a). 'Improving the First-Miss Computation in Set-Associative Instruction Caches'. In: *Proceedings of 20th Euromicro Conference on Real-Time Systems (ECRTS '08)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 341–350. DOI: 10.1109/ECRTS.2008.34.

— (July 2008b). 'Improving the WCET computation time by IPET using control flow graph partitioning'. In: *Proceedings of 8th International Workshop on Worst-Case Execution Time Analysis (WCET '08)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany. DOI: 10.4230/OASIcs.WCET.2008.1670.

Ballabriga, C., H. Cassé and P. Sainrat (Mar. 2008). 'An improved approach for set-associative instruction cache partial analysis'. In: *Proceedings of the 2008 ACM symposium on Applied computing (SAC '08)*. New York, NY, USA: ACM, pp. 360–367. DOI: 10.1145/1363686.1363778.

Ballabriga, C., H. Cassé, C. Rochange and P. Sainrat (Oct. 2011). 'OTAWA: An Open Toolbox for Adaptive WCET Analysis'. In: *Proceedings of 8th International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2010)*. Vol. 6399. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 35–46. DOI: 10.1007/978-3-642-16256-5_6.

Banakar, R., S. Steinke, B.-S. Lee, M. Balakrishnan and P. Marwedel (May 2002). 'Scratchpad Memory: Design Alternative for Cache On-Chip Memory in Embedded Systems'. In: *Proceedings of the 10th international symposium on Hardware/software codesign (CODES '02)*. New York, NY, USA: ACM, pp. 73–78. DOI: 10.1145/774789.774805.

Barre, J., C. Landet, C. Rochange and P. Sainrat (Aug. 2006). 'Modeling Instruction-Level Parallelism for WCET Evaluation'. In: *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2006)*, pp. 61–67. DOI: 10.1109/RTCSA.2006.44.

Barre, J., C. Rochange and P. Sainrat (Feb. 2008). 'A Predictable Simultaneous Multithreading Scheme for Hard Real-Time'. In: *Proceedings of the International Conference on Architecture of Computing Systems (ARCS 2008)*. Vol. 4934. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 161–172. DOI: 10.1007/978-3-540-78153-0_13.

Behrmann, G., A. David and K. Larsen (Sept. 2004). 'A Tutorial on UPPAAL'. In: *Formal Methods for the Design of Real-Time Systems*. Vol. 3185. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 33–35. DOI: 10.1007/978-3-540-30080-9_7.

Benini, L., D. Bruni, M. Chinosi, C. Silvano, V. Zaccaria and R. Zafalon (Sept. 2001). 'A Power Modeling and Estimation Framework for VLIW-based Embedded Systems'. In: *In Proceedings of International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS 2001)*.

Berg, C. (June 2006). 'PLRU Cache Domino Effects'. In: *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET '06)*. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI). DOI: 10.4230/OASIcs.WCET.2006.672.

Berkelaar, M., K. Eikland and P. Notebaert (Aug. 2008). *Open source (Mixed-Integer) Linear Programming system*. http://lpsolve.sourceforge.net/5.5/. Version: 5.5.0.13 Released: 08/05/2008 [Online, last accessed on 3rd March 2012].

Bernat, G., A. Colin and S. M. Petters (Dec. 2002). 'WCET Analysis of Probabilistic Hard Real-Time Systems'. In: *Proceedings of 23rd IEEE International Real-Time Systems Symposium (RTSS 2002)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 279–288. DOI: 10.1109/REAL.2002.1181582.

Bernat, G., A. Burns and M. Newby (Apr. 2005). 'Probabilistic timing analysis: An approach using copulas'. In: *Journal of Embedded Computing – Real-Time Systems (Euromicro RTS-03)* 1 (2), pp. 179–194.

Bhat, B. and F. Mueller (July 2010). 'Making DRAM Refresh Predictable'. In: *Proceedings of 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 145–154. DOI: 10.1109/ECRTS.2010.23.

Bourgade, R., C. Ballabriga, H. Cassé, C. Rochange and P. Sainrat (Oct. 2008). 'Accurate analysis of memory latencies for WCET estimation'. In: *Proceedings of the 16th International Conference on Real-Time and Network Systems (RTNS 2008)*.

Bouyssounouse, B. and J. Sifakis, eds. (2005). *The ARTIST Roadmap for Research and Development*. Embedded Systems Design. Vol. 3436. Lecture Notes in Computer Science. Springer, p. 492. ISBN: 978-3-540-25107-1. DOI: 10.1007/b106761.

Bradford, J. P. and R. Quong (Dec. 1999). 'An empirical study on how program layout affects cache miss rates'. In: *ACM SIGMETRICS Performance Evaluation Review* 27 (3), pp. 28–42. DOI: 10.1145/340242.340326.

Brown, G. G. and R. F. Dell (Jan. 2007). 'Formulating linear and integer linear programs: A rouges' gallery'. In: *INFORMS Transactions on Education (ITE)* 7.2, pp. 153–159. DOI: 10.1287/ited.7.2.153.

Burguiere, C. and C. Rochange (Mar. 2005). 'A contribution to branch prediction modeling in WCET analysis'. In: *Proceedings of the conference on Design, Automation and Test in Europe (DATE '05)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 612–617. DOI: 10.1109/DATE.2005.7.

— (Aug. 2007). 'On the Complexity of Modelling Dynamic Branch Predictors when Computing Worst-Case Execution Times'. In: *Proceedings of the ERCIM/DECOS Workshop on Dependable Embedded Systems*.

Busquets-Mataix, J., J. Serrano, R. Ors, P. Gil and A. Wellings (June 1996). 'Adding instruction cache effect to schedulability analysis of preemptive real-time systems'. In: *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS 1996)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 204–212. DOI: 10.1109/RTTAS.1996.509537.

Busquets-Mataix, J., J. Serrano and A. Wellings (June 1997). 'Hybrid instruction cache partitioning for preemptive real-time systems'. In: *Proceedings of the Ninth Euromicro Workshop on Real-Time Systems*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 56–63. DOI: 10.1109/EMWRTS.1997.613764.

Buttazzo, G. C. (2005). *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Second Edition. Springer Sience+Business Media, Inc. ISBN: 0-387-23137-4.

Bünte, S., M. Zolda and R. Kirner (June 2011a). 'Let's get less optimistic in measurement-based timing analysis'. In: *Proceedings of the 6th IEEE International Symposium on Industrial Embedded Systems (SIES 2011)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 204–212. DOI: 10.1109/SIES.2011.5953663.

Bünte, S., M. Zolda, M. Tautschnig and R. Kirner (Mar. 2011b). 'Improving the Confidence in Measurement-Based Timing Analysis'. In: Los Alamitos, CA, USA: IEEE Computer Society, pp. 144–151. DOI: 10.1109/ISORC.2011.27.

Campoy, A., I. Puaut, A. Ivars and J. Mataix (July 2005). 'Cache contents selection for statically-locked instruction caches: an algorithm comparison'. In: *Proceedings of the 17th Euromicro Conference on Real-Time Systems, 2005 (ECRTS 2005)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 49–56. DOI: 10.1109/ECRTS.2005.34.

Cassé, H. and P. Sainrat (Jan. 2006). 'OTAWA, a framework for experimenting WCET computations'. In: *Proceedings of the 3rd European Congress on Embedded Real-Time Software*.

Chattopadhyay, S. and A. Roychoudhury (Dec. 2009). 'Unified Cache Modeling for WCET Analysis and Layout Optimizations'. In: *Proceedings of the 30th IEEE International Real-Time Systems Symposium (RTSS 2009)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 47–56. DOI: 10.1109/RTSS.2009.20.

— (Dec. 2011). 'Scalable and Precise Refinement of Cache Timing Analysis via Model Checking'. In: *Proceedings of the 32nd IEEE International Real-Time Systems Symposium (RTSS 2011)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 193–203. DOI: 10.1109/RTSS.2011.25.

Chattopadhyay, S., A. Roychoudhury and T. Mitra (June 2010). 'Modeling shared cache and bus in multi-cores for timing analysis'. In: *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems (SCOPES '10)*. New York, NY, USA: ACM, 6:1–6:10. DOI: 10.1145/1811212.1811220.

Chiou, D., P. Jain, L. Rudolph and S. Devadas (June 2000). 'Application-specific memory management for embedded systems using software-controlled caches'. In: *Proceedings of the 37th Annual Design Automation Conference (DAC '00)*. New York, NY, USA: ACM, pp. 416–419. DOI: 10.1145/337292.337523.

Choi, J.-Y., I. Lee and I. Kang (May 1994). 'Timing analysis of superscalar processor programs using ACSR'. In: *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software (RTOSS '94)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 63–67. DOI: 10.1109/RTOSS.1994.292559.

Clark, L., E. Hoffman, J. Miller, M. Biyani, L. Liao, S. Strazdus, M. Morrow, K. Velarde and M. Yarch (Nov. 2001). 'An embedded 32-b microprocessor core for low-power and high-performance applications'. In: *IEEE Journal of Solid-State Circuits* 36.11, pp. 1599–1608. DOI: 10.1109/4.962279.

Colin, A. and I. Puaut (May 2000). 'Worst Case Execution Time Analysis for a Processor with Branch Prediction'. In: *Real-Time Systems* 18 (2), pp. 249–274. DOI: 10.1023/A:1008149332687.

— (June 2001). 'A modular and retargetable framework for tree-based WCET analysis'. In: *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS 2001)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 37–44. DOI: 10.1109/EMRTS.2001.933995.

Cong, J., K. Gururaj, H. Huang, C. Liu, G. Reinman and Y. Zou (Aug. 2011). 'An energy-efficient adaptive hybrid cache'. In: *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED 2011)*, pp. 67–72. DOI: 10.1109/ISLPED.2011.5993609.

Cook, J., I. Kolmanovsky, D. McNamara, E. Nelson and K. Prasad (Feb. 2007). 'Control, Computing and Communications: Technologies for the Twenty-First Century Model T'. In: *Proceedings of the IEEE* 95.2, pp. 334–355. DOI: 10.1109/JPROC.2006.888384.

Cormen, T. H., C. E. Leiserson, R. L. Rivest and C. Stein (2003). *Introduction to Algorithms*. Second Edition. MIT Press Cambridge Massachusetts. ISBN: 0-262-03293-7.

Cullmann, C., C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet and R. Wilhelm (May 2010). 'Predictability considerations in the design of multi-core embedded systems'. In: *Proceedings of the 2010 Conference on Embedded Real-Time Systems and Software (ERTSS 2010)*.

Cullmann, C. and F. Martin (July 2007). 'Data-Flow Based Detection of Loop Bounds'. In: *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis (WCET '07)*. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI). DOI: 10.4230/OASIcs.WCET.2007.1193.

D'Alberto, P., A. Nicolau, A. Veidenbaum and R. Gupta (Feb. 2005). 'Line size adaptivity analysis of parameterized loop nests for direct mapped data cache'. In: *IEEE Transactions on Computers* 54.2, pp. 185–197. DOI: 10.1109/TC.2005.28.

Dalsgaard, A. E., M. C. Olesen, M. Toft, R. R. Hansen and K. G. Larsen (July 2010). 'METAMOC: Modular Execution Time Analysis using Model Checking'. In: *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 113–123. DOI: 10.4230/OASIcs.WCET.2010.113.

d'Ausbourg, B., M. Boyer, E. Noulard and C. Pagetti (Dec. 2011). 'Deterministic Execution on Many-Core Platforms: Application to the SCC'. In: *Proceedings of the 4th Symposium of the Many-core Applications Research Community (MARC)*.

De Michiel, M., A. Bonenfant, H. Cassé and P. Sainrat (Aug. 2008). 'Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation'. In: *Proceedings of 14th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA '08)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 161–166. DOI: 10.1109/RTCSA.2008.53.

De Michiel, M., A. Bonenfant, C. Ballabriga and H. Cassé (Oct. 2010). 'Partial Flow Analysis with oRange'. In: *Proceedings of the 4th International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA 2010)*. Vol. 6416. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 479–482. DOI: 10.1007/978-3-642-16561-0_43.

Deverge, J.-F. and I. Puaut (July 2007a). 'Safe measurement-based WCET estimation'. In: *Proceedings of the 5th International Workshop on Worst-Case Execution Time Analysis (WCET '05)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik. DOI: 10.4230/OASIcs.WCET.2005.808.

— (July 2007b). 'WCET-Directed Dynamic Scratchpad Memory Allocation of Data'. In: *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS '07)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 179–190. DOI: 10.1109/ECRTS.2007.37.

e200z6 (June 2004). *e200z6 PowerPC Core Reference Manual*. Rev. 0. Freescale Semiconductor.

Edwards, S. A. and E. A. Lee (June 2007). 'The case for the precision timed (PRET) machine'. In: *Proceedings of the 44th annual Design Automation Conference (DAC '07)*. New York, NY, USA: ACM, pp. 264–265. DOI: 10.1145/1278480.1278545.

Egger, B., C. Kim, C. Jang, Y. Nam, J. Lee and S. L. Min (Oct. 2006). 'A Dynamic Code Placement Technique for Scratchpad Memory Using Postpass Optimization'. In: *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems (CASES '06)*. New York, NY, USA: ACM, pp. 223–233. DOI: 10.1145/1176760.1176788.

Engblom, J. (2002). 'Processor Pipelines and Static Worst-Case Execution Time Analysis'. PhD thesis. Uppsala University.

Engblom, J. and A. Ermedahl (Nov. 2000). 'Modeling complex flows for worst-case execution time analysis'. In: *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 163–174. DOI: 10.1109/REAL.2000.896006.

Engblom, J., A. Ermedahl, M. Sjödin, J. Gustafsson and H. Hansson (Aug. 2003). 'Worst-case execution-time analysis for embedded real-time systems'. In: *International Journal on Software Tools for Technology Transfer (STTT)* 4 (4), pp. 437–455. DOI: 10.1007/s100090100054.

Ermedahl, A., F. Stappert and J. Engblom (Sept. 2005). 'Clustered worst-case execution-time calculation'. In: *IEEE Transactions on Computers* 54.9, pp. 1104–1122. DOI: 10.1109/TC.2005.139.

Ermedahl, A., C. Sandberg, J. Gustafsson, S. Bygde and B. Lisper (July 2007). 'Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis'. In: *Proceedings of the 7th International Workshop on Worst-Case Execution Time*

Analysis (WCET '07). Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI). DOI: 10.4230/OASIcs.WCET.2007.1194.

Falk, H. and J. Kleinsorge (July 2009). 'Optimal static WCET-aware scratchpad allocation of program code'. In: *Proceedings of the 46th ACM/IEEE Design Automation Conference (DAC '09)*. New York, NY, USA: ACM, pp. 732–737. DOI: 10.1145/1629911.1630101.

Falk, H. and P. Lokuciejewski (Oct. 2010). 'A compiler framework for the reduction of worst-case execution times'. In: *Real-Time Systems* 46 (2), pp. 251–300. DOI: 10.1007/s11241-010-9101-x.

Falk, H., P. Lokuciejewski and H. Theiling (Oct. 2006). 'Design of a WCET-Aware C Compiler'. In: *Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia (ESTMED '06)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 121–126. DOI: 10.1109/ESTMED.2006.321284.

Falk, H., S. Plazar and H. Theiling (Oct. 2007). 'Compile-Time Decided Instruction Cache Locking Using Worst-Case Execution Paths'. In: *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES+ISSS '07)*. New York, NY, USA: ACM, pp. 143–148. DOI: 10.1145/1289816.1289853.

Ferdinand, C. and R. Wilhelm (June 1998). 'On predicting data cache behavior for real-time systems'. In: *In Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '98)*. Vol. 1474. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 16–30. DOI: 10.1007/BFb0057777.

— (Nov. 1999). 'Efficient and Precise Cache Behavior Prediction for Real-Time Systems'. In: *Real-Time Systems* 17 (2), pp. 131–181. DOI: 10.1023/A:1008186323068.

Ferdinand, C., F. Martin and R. Wilhelm (June 1997). 'Applying Compiler Techniques to Cache Behavior Prediction'. In: *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS '97)*, pp. 37–46.

Gebhard, G. and S. Altmeyer (Oct. 2007). 'Optimal task placement to improve cache performance'. In: *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT '07)*. New York, NY, USA: ACM, pp. 259–268. DOI: 10.1145/1289927.1289968.

Gerdes, M., J. Wolf, I. Guliashvili, T. Ungerer, M. Houston, G. Bernat, S. Schnitzler and H. Regler (June 2011). 'Large drilling machine control code – Parallelisation and WCET speedup'. In: *Proceedings of 6th IEEE International Symposium on Industrial Embedded Systems (SIES 2011)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 91–94. DOI: 10.1109/SIES.2011.5953688.

Gloy, N. and M. D. Smith (Sept. 1999). 'Procedure placement using temporal-ordering information'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21 (5), pp. 977–1027. DOI: 10.1145/330249.330254.

Grund, D. and J. Reineke (Aug. 2009). 'Abstract Interpretation of FIFO Replacement'. In: *Proceedings of the 16th International Symposium Static Analysis (SAS 2009)*. Vol. 5673. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 120–136. DOI: 10.1007/978-3-642-03237-0_10.

— (July 2010a). 'Precise and Efficient FIFO-Replacement Analysis Based on Static Phase Detection'. In: *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 155–164. DOI: 10.1109/ECRTS.2010.8.

— (July 2010b). 'Toward Precise PLRU Cache Analysis'. In: *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 23–35. DOI: 10.4230/OASIcs.WCET.2010.23.

Grund, D., J. Reineke and G. Gebhard (Aug. 2009). 'Branch Target Buffers: WCET Analysis Framework and Timing Predictability'. In: *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '09)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 3–12. DOI: 10.1109/RTCSA.2009.8.

Guillon, C., F. Rastello, T. Bidault and F. Bouchez (Sept. 2004). 'Procedure placement using temporal-ordering information: dealing with code size expansion'. In: *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems (CASES '04)*. New York, NY, USA: ACM, pp. 268–279. DOI: 10.1145/1023833.1023870.

Gustafsson, J., A. Ermedahl, C. Sandberg and B. Lisper (Dec. 2006). 'Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution'. In: *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS 2006)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 57–66. DOI: 10.1109/RTSS.2006.12.

Gustafsson, J., A. Betts, A. Ermedahl and B. Lisper (July 2010). 'The Mälardalen WCET Benchmarks: Past, Present And Future'. In: *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 136–146. DOI: 10.4230/OASIcs.WCET.2010.136.

Gustavsson, A., A. Ermedahl, B. Lisper and P. Pettersson (July 2010). 'Towards WCET Analysis of Multicore Architectures Using UPPAAL'. In: *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 101–112. DOI: 10.4230/OASIcs.WCET.2010.101.

Guthaus, M., J. Ringenberg, D. Ernst, T. Austin, T. Mudge and R. Brown (Dec. 2001). 'MiBench: A free, commercially representative embedded benchmark suite'. In: *Proceedings of the 2001 IEEE International Workshop on Workload Characterization (WWC-4 2001)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 3–14. DOI: 10.1109/WWC.2001.990739.

Hansson, A., K. Goossens, M. Bekooij and J. Huisken (Jan. 2009). 'CoMPSoC: A template for composable and predictable multi-processor system on chips'. In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 14 (1), 2:1–2:24. DOI: 10.1145/1455229.1455231.

Hardy, D. and I. Puaut (Dec. 2008). 'WCET Analysis of Multi-level Non-inclusive Set-Associative Instruction Caches'. In: *Proceedings of the 29th IEEE International Real-Time Systems Symposium (RTSS 2008)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 456–466. DOI: 10.1109/RTSS.2008.10.

— (Aug. 2011). 'WCET analysis of instruction cache hierarchies'. In: *Journal of Systems Architecture (JSA)* 57 (7), pp. 677–694. DOI: 10.1016/j.sysarc.2010.08.007.

Hardy, D., T. Piquet and I. Puaut (Dec. 2009). 'Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches'. In: *Proceedings of the 30th IEEE International Real-Time Systems Symposium (RTSS 2009)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 68–77. DOI: 10.1109/RTSS.2009.34.

Hardy, D., B. Lesage and I. Puaut (Dec. 2011). 'Scalable Fixed-Point Free Instruction Cache Analysis'. In: *Proceedings of the 32nd IEEE International Real-Time Systems Symposium (RTSS 2011)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 204–213. DOI: 10.1109/RTSS.2011.26.

Harmon, T., R. Kirner, M. Schoeberl and R. Klefstad (Apr. 2008). 'A Modular Worst-case Execution Time Analysis Tool for Java Processors'. In: *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '08)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 47–57. DOI: 10.1109/RTAS.2008.34.

Healy, C. and D. Whalley (Aug. 2002). 'Automatic detection and exploitation of branch constraints for timing analysis'. In: *IEEE Transactions on Software Engineering* 28.8, pp. 763–781. DOI: 10.1109/TSE.2002.1027799.

Healy, C., D. Whalley and M. Harmon (Dec. 1995). 'Integrating the timing analysis of pipelining and instruction caching'. In: *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 288–297. DOI: 10.1109/REAL.1995.495218.

Healy, C., R. Arnold, F. Mueller, D. Whalley and M. Harmon (Jan. 1999). 'Bounding pipeline and instruction cache performance'. In: *IEEE Transactions on Computers* 48.1, pp. 53–70. DOI: 10.1109/12.743411.

Healy, C., M. Sjödin, V. Rustagi, D. Whalley and R. van Engelen (May 2000). 'Supporting Timing Analysis by Automatic Bounding of Loop Iterations'. In: *Real-Time Systems* 18 (2), pp. 129–156. DOI: 10.1023/A:1008189014032.

Heckmann, R. and C. Ferdinand (2006). *Worst-Case Execution Time Prediction by Static Program Analysis*. Tech. rep. [Online, last accessed on 3rd March 2012]. AbsInt Angewandte Informatik GmbH. URL: http://www.absint.de/aiT_WCET.pdf.

Heckmann, R., M. Langenbach, S. Thesing and R. Wilhelm (July 2003). 'The influence of processor architecture on the design and the results of WCET tools'. In: *Proceedings of the IEEE* 91.7, pp. 1038–1054. DOI: 10.1109/JPROC.2003.814618.

Hennessy, J. and D. Patterson (2006). *Computer Architecture: A Quantitative Approach*. Fourth Edition. Morgan Kaufmann. ISBN: 978-0-12-370490-0.

HighTec. *HighTec EDV-Systeme GmbH Website*. http://www.hightec-rt.com/. [Online, last accessed on 3rd March 2012].

— (2003). *HighTec GNU Toolchain for TriCore - User's Guide*. Preliminary Version 0.81. HighTec EDV-Systeme GmbH.

Hill, M. (Dec. 1988). 'A case for direct-mapped caches'. In: *Computer* 21.12, pp. 25–40. DOI: 10.1109/2.16187.

Holsti, N., J. Gustafsson, G. Bernat, C. Ballabriga, A. Bonenfant, R. Bourgade, H. Cassé, D. Cordes, A. Kadlec, R. Kirner, J. Knoop, P. Lokuciejewski, N. Merriam, M. de Michiel, A. Prantl, B. Rieder, C. Rochange, P. Sainrat and M. Schordan (July 2008). 'WCET 2008 – Report from the Tool Challenge 2008'. In: *Proceedings of the 8th International Workshop on Worst-Case Execution Time Analysis (WCET '08)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik. DOI: 10.4230/OASIcs.WCET.2008.1663.

Holzmann, G. (June 2006). 'The power of 10: rules for developing safety-critical code'. In: *Computer* 39.6, pp. 95–99. DOI: 10.1109/MC.2006.212.

Huber, B. and M. Schoeberl (July 2009). 'Comparison of Implicit Path Enumeration and Model Checking Based WCET Analysis'. In: *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik. DOI: 10.4230/OASIcs.WCET.2009.2281.

Hur, Y., Y. H. Bae, S.-S. Lim, S.-K. Kim, B.-D. Rhee, S. L. Min, C. Y. Park, M. Lee, H. Shin and C. S. Kim (Dec. 1995). 'Worst case timing analysis of RISC processors: R3000/R3010 case study'. In: *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 308–319. DOI: 10.1109/REAL.1995.495220.

Hutton, M., R. Yuan, J. Schleicher, G. Baeckler, S. Cheung, K. K. Chua and H. K. Phoo (Mar. 2006). 'A methodology for FPGA to structured-ASIC synthesis and verification'. In: *Proceedings of the conference on Design, automation and test in Europe: Designers' forum (DATE '06)*. Leuven, Belgium: European Design and Automation Association, pp. 64–69.

Huynh, B. K., L. Ju and A. Roychoudhury (Apr. 2011). 'Scope-Aware Data Cache Analysis for WCET Estimation'. In: *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2011)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 203–212. DOI: 10.1109/RTAS.2011.27.

IEEE Computer Society (Sept. 2011). *IEEE Standard 1666-2011 IEEE Standard for Standard SystemC Language Reference Manual*. IEEE Computer Society.

Jacob, B. (Oct. 1999). 'Hardware/software architectures for real-time caching'. In: *Proceedings of the Second Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99)*, pp. 135–138.

Jacob, B., S. Ng and D. Wang (2007). *Memory systems: cache, DRAM, disk*. Morgan Kaufmann Pub. ISBN: 978-0-12-379751-3.

Janapsatya, A., S. Parameswaran and A. Ignjatovic (Nov. 2004). 'Hardware/software managed scratchpad memory for embedded system'. In: *Proceedings of the 2004 IEEE/ACM International Conference on Computer Aided Design (ICCAD '04)*. Washington, DC, USA: IEEE Computer Society, pp. 370–377. DOI: 10.1109/ICCAD.2004.1382603.

Janapsatya, A., A. Ignjatovic and S. Parameswaran (Jan. 2006a). 'A novel instruction scratchpad memory optimization method based on concomitance metric'. In: *Proceedings of the 2006 Asia and South Pacific Design Automation Conference (ASP-DAC '06)*. Piscataway, NJ, USA: IEEE Press, pp. 612–617. DOI: 10.1145/1118299.1118443.

— (Aug. 2006b). 'Exploiting Statistical Information for Implementation of Instruction Scratchpad Memory in Embedded System'. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14.8, pp. 816–829. DOI: 10.1109/TVLSI.2006.878470.

Johnson, R., D. Pearson and K. Pingali (June 1994). 'The program structure tree: computing control regions in linear time'. In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI '94)*. New York, NY, USA: ACM, pp. 171–185. DOI: 10.1145/178243.178258.

Jungnickel, D. (1999). *Graphs, Networks and Algorithms*. Vol. 5. Algorithms and Computation in Mathematics. Springer Verlag. ISBN: 3-540-63760-5.

Kandemir, M., J. Ramanujam, M. Irwin, N. Vijaykrishnan, I. Kadayif and A. Parikh (June 2001). 'Dynamic management of scratch-pad memory space'. In: *Proceedings of the 38th Design Automation Conference (DAC '01)*. New York, NY, USA: ACM, pp. 690–695. DOI: 10.1145/378239.379049.

Kim, S.-K., S. L. Min and R. Ha (June 1996). 'Efficient worst case timing analysis of data caching'. In: *Proceedings of the 1996 IEEE Real-Time Technology and Applications Symposium (RTAS '96)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 230–240. DOI: 10.1109/RTTAS.1996.509540.

Kirk, D. (Dec. 1989). 'SMART (strategic memory allocation for real-time) cache design'. In: *Proceedings of the 10th Real Time Systems Symposium (RTSS '89)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 229–237. DOI: 10.1109/REAL.1989.63574.

Kirk, D. and J. Strosnider (Dec. 1990). 'SMART (strategic memory allocation for real-time) cache design using the MIPS R3000'. In: *Proceedings of the 11th Real-Time Systems Symposium (RTSS '90)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 322–330. DOI: 10.1109/REAL.1990.128764.

Kirner, R. and M. Schoeberl (June 2007). 'Modeling the Function Cache for Worst-Case Execution Time Analysis'. In: *Proceedings of the 44th annual Design Automation Conference (DAC '07)*, pp. 471–476. DOI: 10.1145/1278480.1278603.

Kirner, R., P. Puschner and I. Wenzel (June 2004). 'Measurement-Based Worst-Case Execution Time Analysis Using Automatic Test-Data Generation'. In: *Proceedings of the 4th Interna-*

tional Workshop on Worst-Case Execution Time Analysis (WCET 2004). Rennes, France: IRISA, pp. 67–70.

Kluge, F., J. Mische, S. Metzlaff, S. Uhrig and T. Ungerer (July 2007). 'Integration of Hard Real-Time and Organic Computing'. In: *ACACES 2007 Poster Abstracts*. Ghent, Belgium: Academia Press.

Kopetz, H. and G. Bauer (Jan. 2003). 'The time-triggered architecture'. In: *Proceedings of the IEEE* 91.1, pp. 112–126. DOI: 10.1109/JPROC.2002.805821.

Kopetz, H. and G. Grünsteidl (June 1993). 'TTP - A time-triggered protocol for fault-tolerant real-time systems'. In: *Proceedings of the Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS-23)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 524–533. DOI: 10.1109/FTCS.1993.627355.

Kuon, I. and J. Rose (Feb. 2007). 'Measuring the Gap Between FPGAs and ASICs'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2, pp. 203–215. DOI: 10.1109/TCAD.2006.884574.

Lampret, D. (Jan. 2011). *OpenRISC 1200 IP Core Specification*. Revision 0.11.

Landet, C. (Oct. 2008). 'Worst Case Execution Time Evaluation for a Simultaneous Multithreaded Processor'. In: *Proceedings of 2nd Junior Researcher Workshop on Real-Time Computing*. Rennes, France: IRISA, pp. 9–12.

— (Dec. 2009). 'Modélisation d'un processeur à exécution simultanée de flots pour le temps réel strict'. PhD thesis. Université de Toulouse.

Larsen, K. G., P. Pettersson and W. Yi (Dec. 1997). 'UPPAAL in a nutshell'. In: *International Journal on Software Tools for Technology Transfer (STTT)* 1 (1), pp. 134–152. DOI: 10.1007/s100090050010.

Lee, E. (July 2005). 'Absolutely Positively on Time: What Would It Take? [Embedded Computing Systems]'. In: *Computer* 38.7, pp. 85–87. DOI: 10.1109/MC.2005.211.

Lee, L. H., B. Moyer and J. Arends (Aug. 1999). 'Instruction fetch energy reduction using loop caches for embedded applications with small tight loops'. In: *Proceeding of the 1999 International Symposium on Low Power Electronics and Design (ISLPED '99)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 267–269. DOI: 10.1109/LPE.1999.145059.

Lesage, B., D. Hardy and I. Puaut (July 2009). 'WCET Analysis of Multi-Level Set-Associative Data Caches'. In: *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET '09)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik. DOI: 10.4230/OASIcs.WCET.2009.2283.

Li, L., L. Gao and J. Xue (Sept. 2005). 'Memory coloring: a compiler approach for scratch-pad memory management'. In: *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT 2005)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 329–338. DOI: 10.1109/PACT.2005.27.

Li, X., A. Roychoudhury and T. Mitra (Dec. 2004). 'Modeling out-of-order processors for software timing analysis'. In: *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS '04)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 92–103. DOI: 10.1109/REAL.2004.33.

Li, X., A. Roychoudhury and T. Mitra (Nov. 2006). 'Modeling out-of-order processors for WCET analysis'. In: *Real-Time Systems* 34 (3), pp. 195–227. DOI: 10.1007/s11241-006-9205-5.

Li, X., Y. Liang, T. Mitra and A. Roychoudhury (Dec. 2007). 'Chronos: A timing analyzer for embedded software'. In: *Science of Computer Programming* 69.1-3, pp. 56–67. DOI: 10.1016/j.scico.2007.01.014.

Li, Y., V. Suhendra, Y. Liang, T. Mitra and A. Roychoudhury (Dec. 2009). 'Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores'. In: *Proceedings of the 30th*

IEEE International Real-Time Systems Symposium (RTSS 2009). Los Alamitos, CA, USA: IEEE Computer Society, pp. 57–67. DOI: 10.1109/RTSS.2009.32.

Li, Y.-T. S. and S. Malik (Nov. 1995). 'Performance analysis of embedded software using implicit path enumeration'. In: *ACM SIGPLAN Notices* 30.11, pp. 88–98. DOI: 10.1145/216633.216666.

Li, Y.-T. S., S. Malik and A. Wolfe (Dec. 1995). 'Efficient microarchitecture modeling and path analysis for real-time software'. In: *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 298–307. DOI: 10.1109/REAL.1995.495219.

— (Dec. 1996). 'Cache modeling for real-time software: beyond direct mapped instruction caches'. In: *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 254–263. DOI: 10.1109/REAL.1996.563722.

Lickly, B., I. Liu, S. Kim, H. Patel, S. Edwards and E. Lee (Oct. 2008). 'Predictable programming on a precision timed architecture'. In: *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems (CASES '08)*. New York, NY, USA: ACM, pp. 137–146. DOI: 10.1145/1450095.1450117.

Lim, S.-S., Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park and C. S. Kim (Dec. 1994). 'An accurate worst case timing analysis technique for RISC processors'. In: *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS '94)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 97–108. DOI: 10.1109/REAL.1994.342726.

Lim, S.-S., Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon and C. S. Kim (July 1995). 'An accurate worst case timing analysis for RISC processors'. In: *IEEE Transactions on Software Engineering* 21.7, pp. 593–604. DOI: 10.1109/32.392980.

Lim, S.-S., J. H. Han, J. Kim and S. L. Min (Dec. 1998). 'A worst case timing analysis technique for multiple-issue machines'. In: *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS '98)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 334–345. DOI: 10.1109/REAL.1998.739765.

Liu, J. W. S. (2000). *Real-Time Systems*. Prentice-Hall, Inc. ISBN: 0-13-099651-3.

Lokuciejewski, P., H. Falk and P. Marwedel (July 2008a). 'WCET-driven Cache-based Procedure Positioning Optimizations'. In: *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS '08)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 321–330. DOI: 10.1109/ECRTS.2008.20.

Lokuciejewski, P., H. Falk, P. Marwedel and H. Theiling (Mar. 2008b). 'WCET-driven, code-size critical procedure cloning'. In: *Proceedings of the 11th international workshop on Software & compilers for embedded systems (SCOPES '08)*. New York, NY, USA: ACM, pp. 21–30.

Luenberger, D. G. and Y. Ye (2008). *Linear and Nonlinear Programming*. Third Edition. Springer New York. ISBN: 978-0-387-74502-2.

Lundqvist, T. and P. Stenström (June 1998). 'Integrating path and timing analysis using instruction-level simulation techniques'. In: *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '08)*. Vol. 1474. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 1–15. DOI: 10.1007/BFb0057776.

— (Dec. 1999). 'Timing anomalies in dynamically scheduled microprocessors'. In: *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS '99)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 12–21. DOI: 10.1109/REAL.1999.818824.

Lv, M., W. Yi, N. Guan and G. Yu (Dec. 2010a). 'Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software'. In: *Proceedings of the 31st IEEE International Real-Time Systems Symposium (RTSS 2010)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 339–349. DOI: 10.1109/RTSS.2010.30.

Lv, M., N. Guan, W. Yi, Q. Deng and G. Yu (Apr. 2010b). 'Efficient Instruction Cache Analysis with Model Checking'. In: *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2010) – Work-in-Progress Session*, pp. 33–36.

Lv, M., N. Guan, Q. Deng, G. Yu and W. Yi (Oct. 2011). 'McAiT – A Timing Analyzer for Multicore Real-Time Software'. In: *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis (ATVA 2011)*. Vol. 6996. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 414–417. DOI: 10.1007/978-3-642-24372-1_29.

Mälardalen Real-Time Research Center (MRTC). *WCET Benchmark Suite*. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html. [Online, last accessed on 3rd March 2012].

Malik, A., B. Moyer and D. Cermak (July 2000). 'A low power unified cache architecture providing power and performance flexibility'. In: *Proceedings of the 2000 International Symposium on Low Power Electronics and Design – Poster Session (ISLPED '00)*. New York, NY, USA: ACM, pp. 241–243. DOI: 10.1145/344166.344610.

Martin, F. M., M. Alt, R. Wilhelm and C. Ferdinand (Apr. 1998). 'Analysis of Loops'. In: *Proceedings of the 7th International Conference on Compiler Construction, (CC '98)*. Vol. 1383. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 80–94. DOI: 10.1007/BFb0026424.

Marwedel, P. (2007). *Eingebettete Systeme*. Springer Verlag. ISBN: 978-3-540-34048-5.

McFarling, S. (Apr. 1989). 'Program optimization for instruction caches'. In: *Proceedings of the third international conference on Architectural support for programming languages and operating systems (ASPLOS-III)*. New York, NY, USA: ACM, pp. 183–191. DOI: 10.1145/70082.68200.

MERASA (Oct. 2009a). *Deliverable 2.4: Detailed SystemC Multi-Core Simulator (Low-Level MERASA Simulator)*. http://www.merasa.org/downloads/Deliverable_2_4.pdf. [Online, last accessed on 3rd March 2012].

— (Oct. 2009b). *Deliverable 3.3: Model of the SystemC Model of the MERASA Multi-Core Processor*. http://www.merasa.org/downloads/Deliverable_3_3.pdf. [Online, last accessed on 3rd March 2012].

Metzlaff, S. and T. Ungerer (June 2012a). 'Impact of Instruction Cache and Different Instruction Scratchpads on the WCET Estimate'. In: *Proceedings of the 9th IEEE International Conference on Embedded Software and Systems (ICESS-2012)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 1442–1449. DOI: 10.1109/HPCC.2012.211.

— (July 2012b). 'Replacement Policies for a Function-based Instruction Memory: A Quantification of the Impact on Hardware Complexity and WCET Estimates'. In: *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 112–121. DOI: 10.1109/ECRTS.2012.22.

Metzlaff, S., S. Uhrig, J. Mische and T. Ungerer (Oct. 2008). 'Predictable Dynamic Instruction Scratchpad for Simultaneous Multithreaded Processors'. In: *Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture (MEDEA '08)*. New York, NY, USA: ACM, pp. 38–45. DOI: 10.1145/1509084.1509090.

Metzlaff, S., I. Guliashvili, S. Uhrig and T. Ungerer (Feb. 2011a). 'A Dynamic Instruction Scratchpad Memory for Embedded Processors Managed by Hardware'. In: *Proceedings of the 24th International Conference on Architecture of Computing Systems (ARCS 2011)*. Vol. 6566. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 122–134. DOI: 10.1007/978-3-642-19137-4_11.

Metzlaff, S., J. Mische and T. Ungerer (Nov. 2011b). 'A Real-Time Capable Many-Core Model'. In: *Proceedings of 32nd IEEE Real-Time Systems Symposium: Work-in-Progress Session*, pp. 21–24.

Metzner, A. (July 2004). 'Why Model Checking Can Improve WCET Analysis'. In: *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004)*. Vol. 3114. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 298–301. DOI: 10.1007/978-3-540-27813-9_26.

Mezzetti, E., N. Holsti, A. Colin, G. Bernat and T. Vardanega (Oct. 2008). 'Attacking the Sources of Unpredictability in the Instruction Cache Behavior'. In: *Proceedings of the 16th International Conference on Real-Time and Network Systems (RTNS 2008)*.

Milenkovic, A., M. Milenkovic and N. Barnes (Mar. 2003). 'A performance evaluation of memory hierarchy in embedded systems'. In: *Proceedings of the 35th Southeastern Symposium on System Theory (SSST 2003)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 427–431. DOI: 10.1109/SSST.2003.1194606.

Mische, J. (Dec. 2011). 'Echtzeitfähige Ablaufplanung für simultan mehrfädige Prozessoren'. PhD thesis. Universität Augsburg.

Mische, J., S. Uhrig, F. Kluge and T. Ungerer (Oct. 2008). 'Exploiting Spare Resources of In-order SMT Processors Executing Hard Real-time Threads'. In: *Proceedings of the 26th IEEE International Conference on Computer Design 2008 (ICCD '08)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 371–376. DOI: 10.1109/ICCD.2008.4751887.

— (Jan. 2009). 'IPC Control for Multiple Real-Time Threads on an In-Order SMT Processor'. In: *Proceedings of the Fourth International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2009)*. Vol. 5409. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 125–139. DOI: 10.1007/978-3-540-92990-1_11.

Mische, J., I. Guliashvili, S. Uhrig and T. Ungerer (Feb. 2010a). 'How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT'. In: *Proceedings of the 23rd International Conference on Architecture of Computing Systems (ARCS 2010)*. Vol. 5974. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 2–14. DOI: 10.1007/978-3-642-11950-7_2.

Mische, J., S. Uhrig, F. Kluge and T. Ungerer (Aug. 2010b). 'Using SMT to Hide Context Switch Times of Large Real-Time Tasksets'. In: *Proceedings of the IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2010)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 255–264. DOI: 10.1109/RTCSA.2010.33.

Montanaro, J., R. Witek, K. Anne, A. Black, E. Cooper, D. Dobberpuhl, P. Donahue, J. Eno, W. Hoeppner, D. Kruckemyer, T. Lee, P. Lin, L. Madden, D. Murray, M. Pearce, S. Santhanam, K. Snyder, R. Stehpany and S. Thierauf (Nov. 1996). 'A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor'. In: *IEEE Journal of Solid-State Circuits* 31.11, pp. 1703–1714. DOI: 10.1109/JSSC.1996.542315.

Moshnyaga, V. G. (Feb. 2001). 'Reducing cache engery through dual voltage supply'. In: *Proceedings of the 6th Asia and South Pacific Design Automation Conference (ASP-DAC '01)*. New York, NY, USA: ACM, pp. 302–305. DOI: 10.1145/370155.370362.

MPC5554 (Apr. 2007). *MPC5553/5554 Microcontroller Reference Manual*. Rev. 4.0. Freescale Semiconductor.

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. San Diego: Academic Press. ISBN: 1-55860-320-4.

Mueller, F. (July 1994). 'Static cache simulation and its applications'. PhD thesis. Florida State University.

— (Nov. 1995). 'Compiler support for software-based cache partitioning'. In: *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems (LCTES/LCT-RTS '95)*. New York, NY, USA: ACM, pp. 125–133. DOI: 10.1145/216636.216677.

Mueller, F. (June 1997). 'Timing Predictions for Multi-Level Caches'. In: *In ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems (LCT-RTS '97)*, pp. 29–36.

— (May 2000). 'Timing Analysis for Instruction Caches'. In: *Real-Time Systems* 18 (2), pp. 217–247. DOI: 10.1023/A:1008145215849.

Mueller, F. and D. Whalley (Sept. 1994). 'Efficient on-the-fly analysis of program behavior and static cache simulation'. In: *Proceedings of the First International Static Analysis Symposium (SAS '04)*. Vol. 864. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 101–115. DOI: 10.1007/3-540-58485-4_35.

Mueller, F., D. B. Whalley and M. Harmon (June 1994). 'Predicting Instruction Cache Behavior'. In: *Proceedings of the ACM SIGPLAN 1994 Workshop on Language, Compiler, and Tool Support for Real-Time Systems (LCT-RTS '94)*.

Myers, E. M. (Jan. 1981). 'A precise inter-procedural data flow algorithm'. In: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '81)*. New York, NY, USA: ACM, pp. 219–230. DOI: 10.1145/567532.567556.

Nielson, F., H. Nielson and C. Hankin (1999). *Principles of program analysis*. Springer Verlag. ISBN: 3-540-65410-0.

NIST (1995). *National Institute of Standards and Technology, SECURE HASH STANDARD - Federal Information Processing Standards Publication 180-1*. http://www.itl.nist.gov/fipspubs/fip180-1.htm. [Online, last accessed on 3rd March 2012].

OpenCores (Apr. 2006). *OpenRISC 1000 Architecture Manual*. Revision 1.3.

Pagiamtzis, K. and A. Sheikholeslami (Mar. 2006). 'Content-addressable memory (CAM) circuits and architectures: a tutorial and survey'. In: *IEEE Journal of Solid-State Circuits* 41 (3), pp. 712–727. DOI: 10.1109/JSSC.2005.864128.

Panda, P. R., F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle and P. G. Kjeldsberg (Apr. 2001). 'Data and memory optimization techniques for embedded systems'. In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 6 (2), pp. 149–206. DOI: 10.1145/375977.375978.

Panda, P. R., N. D. Dutt and A. Nicolau (Mar. 1997). 'Efficient utilization of scratch-pad memory in embedded processor applications'. In: *Proceedings of the 1997 European Conference on Design and Test (ED&TC '97)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 7–11. DOI: 10.1109/EDTC.1997.582323.

— (Jan. 1999). 'Local memory exploration and optimization in embedded systems'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18 (1), pp. 3–13. DOI: 10.1109/43.739054.

Paolieri, M., E. Quiñones, F. Cazorla and M. Valero (Dec. 2009a). 'An Analyzable Memory Controller for Hard Real-Time CMPs'. In: *IEEE Embedded Systems Letters* 1 (4), pp. 86–90. DOI: 10.1109/LES.2010.2041634.

Paolieri, M., E. Quiñones, F. J. Cazorla, G. Bernat and M. Valero (June 2009b). 'Hardware support for WCET analysis of hard real-time multicore systems'. In: *Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09)*. New York, NY, USA: ACM, pp. 57–68. DOI: 10.1145/1555754.1555764.

Paolieri, M., J. Mische, S. Metzlaff, M. Gerdes, E. Quiñones, S. Uhrig, T. Ungerer and F. J. Cazorla. (2012). 'A Hard Real-Time Capable Multi-Core SMT Processor'. In: *ACM Transactions on Embedded Computing Systems (TECS)*. To be published.

Park, C. and A. Shaw (May 1991). 'Experiments with a program timing tool based on source-level timing schema'. In: *Computer* 24 (5), pp. 48–57. DOI: 10.1109/2.76286.

Park, S., H.-w. Park and S. Ha (Apr. 2007). 'A novel technique to use scratch-pad memory for stack management'. In: *Proceedings of the Design, Automation and Test in Europe Conference*

& Exhibition (DATE '07). San Jose, CA, USA: EDA Consortium, pp. 1478–1483. DOI: 10.1109/DATE.2007.364509.

Patil, K., K. Seth and F. Mueller (June 2004). 'Compositional static instruction cache simulation'. In: *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES '04)*. New York, NY, USA: ACM, pp. 136–145. DOI: 10.1145/997163.997183.

Patterson, D. and J. Hennessy (2005). *Computer Organization and Design: The Hardware/Software Interface*. Third Edition. Morgan Kaufmann. ISBN: 1-55860-604-1.

Pitter, C. and M. Schoeberl (Sept. 2007). 'Towards a Java multiprocessor'. In: *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES '07)*. New York, NY, USA: ACM, pp. 144–151. DOI: 10.1145/1288940.1288962.

— (Aug. 2010). 'A real-time Java chip-multiprocessor'. In: *ACM Transactions on Embedded Computing Systems (TECS)* 10 (1), 9:1–9:34. DOI: 10.1145/1814539.1814548.

Plazar, S., P. Lokuciejewski and P. Marwedel (July 2009). 'WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems'. In: *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET '09)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Germany. DOI: 10.4230/OASIcs.WCET.2009.2286.

— (May 2010). 'WCET-driven Cache-aware Memory Content Selection'. In: *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2010)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 107–114. DOI: 10.1109/ISORC.2010.36.

Poletti, F., P. Marchal, D. Atienza, L. Benini, F. Catthoor and J. M. Mendias (June 2004). 'An integrated hardware/software approach for run-time scratchpad management'. In: *Proceedings of the 41st annual Design Automation Conference (DAC '04)*. New York, NY, USA: ACM, pp. 238–243. DOI: 10.1145/996566.996634.

Poovey, J., T. Conte, M. Levy and S. Gal-On (Oct. 2009). 'A Benchmark Characterization of the EEMBC Benchmark Suite'. In: *IEEE Micro* 29 (5), pp. 18–29. DOI: 10.1109/MM.2009.74.

Preußer, T., M. Zabel and R. Spallek (Sept. 2007). 'Bump-pointer method caching for embedded Java processors'. In: *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES '07)*. New York, NY, USA: ACM, p. 210. DOI: 10.1145/1288940.1288970.

Puaut, I. (July 2006). 'WCET-centric software-controlled instruction caches for hard real-time systems'. In: *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS 2006)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 217–226. DOI: 10.1109/ECRTS.2006.32.

Puaut, I. and D. Decotigny (Dec. 2002). 'Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems'. In: *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS 2002)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 114–123. DOI: 10.1109/REAL.2002.1181567.

Puaut, I. and C. Pais (Oct. 2006). *Scratchpad memories vs locked caches in hard real-time systems: a qualitative and quantitative comparison*. Tech. rep. PI 1818. IRISA.

— (Apr. 2007). 'Scratchpad Memories vs Locked Caches in Hard Real-Time Systems: a Quantitative Comparison'. In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE '07)*. San Jose, CA, USA: EDA Consortium, pp. 1484–1489. DOI: 10.1109/DATE.2007.364510.

Puschner, P. P. and A. V. Schedl (July 1997). 'Computing Maximum Task Execution Times – A Graph-Based Approach'. In: *Real-Time Systems* 13.1, pp. 67–91. DOI: 10.1023/A:1007905003094.

Qadri, M., H. Gujarathi and K. McDonald-Maier (Oct. 2009). 'Low Power Processor Architectures and Contemporary Techniques for Power Optimization – A Review'. In: *Journal of Computers* 4.10, pp. 927–942. DOI: 10.4304/jcp.4.10.927-942.

Quiñones, E., E. Berger, G. Bernat and F. Cazorla (July 2009). 'Using Randomized Caches in Probabilistic Real-Time Systems'. In: *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS '09)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 129–138. DOI: 10.1109/ECRTS.2009.30.

Rakib, A., O. Parshin, S. Thesing and R. Wilhelm (Nov. 2004). 'Component-Wise Instruction-Cache Behavior Prediction'. In: *Proceedings of the Second International Conference on Automated Technology for Verification and Analysis (ATVA 2004)*. Vol. 3299. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 211–229. DOI: 10.1007/978-3-540-30476-0_20.

RapiTime. *Rapita Systems Ltd. RapiTime Explained – White Paper*. http://www.rapitasystems.com/system/files/RapiTime_Explained_White_Paper.pdf. [Online, last accessed on 3rd March 2012]. Rapita Systems Ltd.

Ratsiambahotra, T., H. Cassé and P. Sainrat (July 2009). 'A versatile generator of instruction set simulators and disassemblers'. In: *Proceedings of the International Symposium on Performance Evaluation of Computer Telecommunication Systems (SPECTS 2009)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 65–72.

Ravindran, R. A., P. D. Nagarkar, G. S. Dasika, E. D. Marsman, R. M. Senger, S. A. Mahlke and R. B. Brown (Mar. 2005). 'Compiler Managed Dynamic Instruction Placement in a Low-Power Code Cache'. In: *Proceedings of the international symposium on Code generation and optimization (CGO '05)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 179–190. DOI: 10.1109/CGO.2005.13.

Reineke, J. (Nov. 2008). 'Caches in WCET Analysis'. PhD thesis. Universität des Saarlandes.

Reineke, J. and D. Grund (June 2008). 'Relative competitive analysis of cache replacement policies'. In: *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '08)*. New York, NY, USA: ACM, pp. 51–60. DOI: 10.1145/1375657.1375665.

Reineke, J., D. Grund, C. Berg and R. Wilhelm (Nov. 2007). 'Timing Predictability of Cache Replacement Policies'. In: *Real-Time Systems* 37.2, pp. 99–122. DOI: 10.1007/s11241-007-9032-3.

Reineke, J., I. Liu, H. D. Patel, S. Kim and E. A. Lee (Oct. 2011). 'PRET DRAM controller: Bank privatization for predictability and temporal isolation'. In: *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '11)*. New York, NY, USA: ACM, pp. 99 –108. DOI: 10.1145/2039370.2039388.

Reinman, G. and N. Jouppi (Feb. 2000). *CACTI 2.0: An Integrated Cache Timing and Power Model*. Tech. rep. Compaq Western Research Laboratory.

Rochange, C. and P. Sainrat (2009). 'A Context-Parameterized Model for Static Analysis of Execution Times'. In: *Transactions on High-Performance Embedded Architectures and Compilers II*. Lecture Notes in Computer Science 5470, pp. 222–241. DOI: 10.1007/978-3-642-00904-4_12.

Sangiovanni-Vincentelli, A. and M. Di Natale (Oct. 2007). 'Embedded System Design for Automotive Applications'. In: *Computer* 40.10, pp. 42–51. DOI: 10.1109/MC.2007.344.

Schneider, J. (Nov. 2000). 'Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems'. In: *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 195–204. DOI: 10.1109/REAL.2000.896009.

Schneider, J. and C. Ferdinand (May 1999). 'Pipeline behavior prediction for superscalar processors by abstract interpretation'. In: *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '99)*. New York, NY, USA: ACM, pp. 35–44. DOI: 10.1145/314403.314432.

Schoeberl, M. (Oct. 2004). 'A Time Predictable Instruction Cache for a Java Processor'. In: *Proceedings of the Workshop On the Move to Meaningful Internet Systems 2004 (OTM 2004)*. Vol. 3292. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 371–382. DOI: 10.1007/978-3-540-30470-8_52.

— (Jan. 2005). 'JOP: A Java Optimized Processor for Embedded Real-Time Systems'. PhD thesis. Vienna University of Technology.

— (Oct. 2007). 'SimpCon - a Simple and Efficient SoC Interconnect'. In: *Proceedings of the 15th Austrian Workhop on Microelectronics (Austrochip 2007)*. IEEE Austria Section / TU Graz, pp. 153–161.

— (Feb. 2008). 'A Java processor architecture for embedded real-time systems'. In: *Journal of Systems Architecture (JSA)* 54.1-2, pp. 265–286. DOI: 10.1016/j.sysarc.2007.06.001.

Schoeberl, M. and R. Pedersen (Oct. 2006). 'WCET analysis for a Java processor'. In: *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems (JTRES '06)*. New York, NY, USA: ACM, pp. 202–211. DOI: 10.1145/1167999.1168033.

Schoeberl, M., W. Puffitsch, R. U. Pedersen and B. Huber (May 2010). 'Worst-case execution time analysis for a Java processor'. In: *Software: Practice and Experience* 40.6, pp. 507–542. DOI: 10.1002/spe.968.

Sepp, A. (2009). 'WCET-basierte Optimierung der Befehlsabbildung für On-Chip-Speicher'. Diploma Thesis. Universität Augsburg.

Shaw, A. (July 1989). 'Reasoning about time in higher-level language software'. In: *IEEE Transactions on Software Engineering* 15 (7), pp. 875–889. DOI: 10.1109/32.29487.

Sheu, T.-L., Y.-B. Shieh and W. Lin (Nov. 1990). 'The selection of optimal cache lines for microprocessor-based controllers'. In: *Proceedings of the 23rd Annual Workshop on Microprogramming and Microarchitecture (MICRO 23)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 183–192. DOI: 10.1109/MICRO.1990.151441.

Shin, K. and P. Ramanathan (Jan. 1994). 'Real-time computing: a new discipline of computer science and engineering'. In: *Proceedings of the IEEE* 82 (1), pp. 6–24. DOI: 10.1109/5.259423.

Shiue, W.-T. and C. Chakrabarti (June 1999). 'Memory exploration for low power, embedded systems'. In: *Proceedings of the 36th Design Automation Conference (DAC '99)*. New York, NY, USA: ACM, pp. 140–145. DOI: 10.1109/DAC.1999.781299.

Smith, A. J. (Sept. 1987). 'Line (Block) Size Choice for CPU Cache Memories'. In: *IEEE Transactions on Computers* C-36 (9), pp. 1063–1075. DOI: 10.1109/TC.1987.5009537.

Srinivasan, S., V. Cuppu and B. Jacob (Nov. 2001). 'Transparent data-memory organizations for digital signal processors'. In: *Proceedings of the 2001 international conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '01)*. New York, NY, USA: ACM, pp. 44–48. DOI: 10.1145/502217.502224.

Stappert, F., A. Ermedahl and J. Engblom (Nov. 2001). 'Efficient longest executable path search for programs with complex flows and pipeline effects'. In: *Proceedings of the 2001 international conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '01)*. New York, NY, USA: ACM, pp. 132–140. DOI: 10.1145/502217.502240.

Staschulat, J. (2006). 'Instruction and Data Cache Timing Analysis in Fixed-Priority Preemptive Real-Time Systems'. PhD thesis. Technische Universität Braunschweig.

Steinke, S., L. Wehmeyer, B.-S. Lee and P. Marwedel (Mar. 2002a). 'Assigning program and data objects to scratchpad for energy reduction'. In: *Proceedings of the 2002 Design, Automation*

and Test in Europe Conference and Exhibition (DATE 2002). Los Alamitos, CA, USA: IEEE Computer Society, pp. 409–415. DOI: 10.1109/DATE.2002.998306.

Steinke, S., N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan and P. Marwedel (Oct. 2002b). 'Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory'. In: *Proceedings of the 15 th International Symposium on System Synthesis (ISSS '02)*. New York, NY, USA: ACM, pp. 213–218. DOI: 10.1145/581199.581247.

Suhendra, V. and T. Mitra (June 2008). 'Exploring locking & partitioning for predictable shared caches on multi-cores'. In: *Proceedings of the 45th annual Design Automation Conference (DAC '08)*. New York, NY, USA: ACM, pp. 300–303. DOI: 10.1145/1391469.1391545.

Suhendra, V., T. Mitra, A. Roychoudhury and T. Chen (Dec. 2005). 'WCET Centric Data Allocation to Scratchpad Memory'. In: *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS 2005)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 223–232. DOI: 10.1109/RTSS.2005.45.

Suhendra, V., T. Mitra, A. Roychoudhury and T. Chen (July 2006). 'Efficient detection and exploitation of infeasible paths for software timing analysis'. In: *Proceedings of the 43rd annual Design Automation Conference (DAC '06)*. New York, NY, USA: ACM, pp. 358–363. DOI: 10.1109/DAC.2006.229300.

Theiling, H. (Dec. 2000). 'Extracting safe and precise control flow from binaries'. In: *Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications (RTCSA 2000)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 23–30. DOI: 10.1109/RTCSA.2000.896367.

Theiling, H., C. Ferdinand and R. Wilhelm (May 2000). 'Fast and precise WCET prediction by separated cache and path analyses'. In: *Real-Time Systems* 18 (2), pp. 157–179. DOI: 10.1023/A:1008141130870.

Thesing, S. (July 2004). 'Safe and Precide WCET Determination by Abstract Interpretation of Pipeline Models'. PhD thesis. Universtität des Saarlandes.

Thiele, L. and R. Wilhelm (Nov. 2004). 'Design for Timing Predictability'. In: *Real-Time Systems* 28 (2), pp. 157–177. DOI: 10.1023/B:TIME.0000045316.66276.6e.

TriCore (May 2002). *TriCore 1.3 32-bit Unified Processor Core Architecture Overview Handbook*. V1.3.3. Infineon Technologies AG.

— (Dec. 2003). *TriCore 32-bit Unified Processor Compiler Writer's Guide*. V1.4. Infineon Technologies AG.

— (June 2004). *TriCore 1 Guidelines for Cache Management*. V1.1. Infineon Technologies AG.

— (Nov. 2007a). *TriCore 1 User's Manual - Volume 1 Core Architecture*. V1.3.8. Infineon Technologies AG.

— (Nov. 2007b). *TriCore 1 User's Manual - Volume 2 Instruction Set*. V1.3.8. Infineon Technologies AG.

Udayakumaran, S. and R. Barua (Nov. 2003). 'Compiler-decided dynamic memory allocation for scratch-pad based embedded systems'. In: *Proceedings of the 2003 international conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '03)*. New York, NY, USA: ACM, pp. 276–286. DOI: 10.1145/951710.951747.

Udayakumaran, S., A. Dominguez and R. Barua (May 2006). 'Dynamic allocation for scratch-pad memory using compile-time decisions'. In: *ACM Transactions on Embedded Computing Systems (TECS)* 5 (2), pp. 472–511. DOI: 10.1145/1151074.1151085.

Ungerer, T., F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quiñones, M. Gerdes, M. Paolieri, J. Wolf, H. Cassé, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff and J. Mische (Oct. 2010). 'Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability'. In: *IEEE Micro* 30 (5), pp. 66–75. DOI: 10.1109/MM.2010.78.

Vander Aa, T., M. Jayapala, F. Barat, G. Deconinck, R. Lauwereins, H. Corporaal and F. Catthoor (Sept. 2003). 'Instruction Buffering Exploration for Low Energy Embedded Processors'. In: *Proceedings of the 13th International Workshop on Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation (PATMOS 2003)*. Vol. 2799. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 409–419. DOI: 10.1007/978-3-540-39762-5_47.

— (2005). 'Instruction Buffering Exploration for Low Energy Embedded Processors'. In: *Journal of Embedded Computing* 1.3, pp. 341–351.

Vera, X., B. Lisper and J. Xue (Dec. 2003). 'Data Caches in Multitasking Hard Real-Time Systems'. In: *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS 2003)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 154–165. DOI: 10.1109/REAL.2003.1253263.

— (Dec. 2007). 'Data cache locking for tight timing calculations'. In: *ACM Transactions on Embedded Computing Systems (TECS)* 7 (1), 4:1–4:38. DOI: 10.1145/1324969.1324973.

Verma, M. and P. Marwedel (Aug. 2006). 'Overlay techniques for scratchpad memories in low power embedded processors'. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14 (8), pp. 802–815. DOI: 10.1109/TVLSI.2006.878469.

Verma, M., L. Wehmeyer and P. Marwedel (Sept. 2004). 'Dynamic overlay of scratchpad memory for energy minimization'. In: *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '04)*. New York, NY, USA: ACM, pp. 104–109. DOI: 10.1145/1016720.1016748.

Von Hanxleden, R., N. Holsti, B. Lisper, E. Ploedereder, R. Wilhelm, A. Bonenfant, H. Cassé, S. Bünte, W. Fellger, S. Gepperth, J. Gustafsson, B. Huber, N. M. Islam, D. Kästner, R. Kirner, L. Kovács, F. Krause, M. de Michiel, M. C. Olesen, A. Prantl, W. Puffitsch, C. Rochange, M. Schoeberl, S. Wegener, M. Zolda and J. Zwirchmayr (July 2011). 'WCET Tool Challenge 2011: Report'. In: *Proceedings of the 11th International Workshop on Worst-Case Execution Time Analysis (WCET 2011)*. Vienna, Austria: OCG.

Wehmeyer, L. and P. Marwedel (June 2004). 'Influence of Onchip Scratchpad Memories on WCET Prediction'. In: *Proceedings of the 4th International Workshop on Worst-Case Execution Time Analysis (WCET 2004)*. Rennes, France: IRISA, pp. 29–32.

— (Mar. 2005). 'Influence of Memory Hierarchies on Predictability for Time Constrained Embedded Software'. In: *Proceedings of the conference on Design, Automation and Test in Europe Conference and Exhibition (DATE '05)*. Washington, DC, USA: IEEE Computer Society, pp. 600–605. DOI: 10.1109/DATE.2005.183.

— (2006). *Fast, Efficient and Predictable Memory Accesses: Optimization Algorithms for Memory Architecture Aware Compilation*. Springer. ISBN: 1-4020-4821-1.

Wenzel, I., R. Kirner, B. Rieder and P. Puschner (May 2005a). 'Measurement-based worst-case execution time analysis'. In: *Proceedings of the Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2005)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 7–10. DOI: 10.1109/SEUS.2005.12.

Wenzel, I., R. Kirner, P. Puschner and B. Rieder (Sept. 2005b). 'Principles of timing anomalies in superscalar processors'. In: *Proceedings of the Fifth International Conference on Quality Software (QSIC 2005)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 295–303. DOI: 10.1109/QSIC.2005.49.

Wenzel, I., R. Kirner, B. Rieder and P. Puschner (Oct. 2009). 'Measurement-Based Timing Analysis'. In: *Proceedings of the Third International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008)*. Vol. 17. Communications in Computer and Information Science. Springer Berlin / Heidelberg, pp. 430–444. DOI: 10.1007/978-3-540-88479-8_30.

White, R., F. Mueller, C. Healy, D. Whalley and M. Harmon (June 1997). 'Timing analysis for data caches and set-associative caches'. In: *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS '97)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 192–202. DOI: 10.1109/RTTAS.1997.601358.

Whitham, J. and N. Audsley (Apr. 2008). 'Using Trace Scratchpads to Reduce Execution Times in Predictable Real-Time Architectures'. In: *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '08)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 305–316. DOI: 10.1109/RTAS.2008.11.

— (Oct. 2009). 'Implementing time-predictable load and store operations'. In: *Proceedings of the seventh ACM international conference on Embedded software (EMSOFT '09)*. New York, NY, USA: ACM, pp. 265–274. DOI: 10.1145/1629335.1629371.

Wilhelm, R., D. Grund, J. Reineke, M. Schlickling, M. Pister and C. Ferdinand (July 2009). 'Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28 (7), pp. 966–978. DOI: 10.1109/TCAD.2009.2013287.

Wilhelm, R., J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat and P. Stenström (Apr. 2008). 'The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools'. In: *ACM Transactions on Embedded Computing Systems (TECS)* 7 (3), pp. 1–53. DOI: 10.1145/1347375.1347389.

Wolf, J., M. Gerdes, F. Kluge, S. Uhrig, J. Mische, S. Metzlaff, C. Rochange, H. Cassé, P. Sainrat and T. Ungerer (May 2010). 'RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-core Processor'. In: *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2010)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 193–201. DOI: 10.1109/ISORC.2010.31.

— (Nov. 2011). 'RTOS Support for Execution of Parallelized Hard Real-Time Tasks on the MERASA Multi-core Processor'. In: *Computer Systems Science & Engineering* 26.6.

Wong, H., V. Betz and J. Rose (Mar. 2011). 'Comparing FPGA vs. custom CMOS and the impact on processor microarchitecture'. In: *Proceedings of the 19th ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA '11)*. New York, NY, USA: ACM, pp. 5–14. DOI: 10.1145/1950413.1950419.

Yan, J. and W. Zhang (Apr. 2008). 'WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches'. In: *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '08)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 80–89. DOI: 10.1109/RTAS.2008.6.

Yoon, M.-K., J.-E. Kim and L. Sha (Dec. 2011). 'Optimizing Tunable WCET with Shared Resource Allocation and Arbitration in Hard Real-Time Multicore Systems'. In: *Proceedings of the 32nd IEEE International Real-Time Systems Symposium (RTSS 2011)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 227–238. DOI: 10.1109/RTSS.2011.28.

Zhang, C., F. Vahid and W. Najjar (May 2005). 'A highly configurable cache for low energy embedded systems'. In: *ACM Transactions on Embedded Computing Systems (TECS)* 4 (2), pp. 363–387. DOI: 10.1145/1067915.1067921.

Zhang, N., A. Burns and M. Nicholson (Oct. 1993). 'Pipelined processors and worst case execution times'. In: *Real-Time Systems* 5 (4), pp. 319–343. DOI: 10.1007/BF01088834.

Zhao, W., D. Whalley, C. Healy and F. Mueller (Dec. 2004). 'WCET code positioning'. In: *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 81–91. DOI: 10.1109/REAL.2004.55.

— (Dec. 2005). ‘Improving WCET by applying a WC code-positioning optimization’. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 2 (4), pp. 335–365. DOI: 10.1145/1113841.1113842.

List of Figures

List of Tables

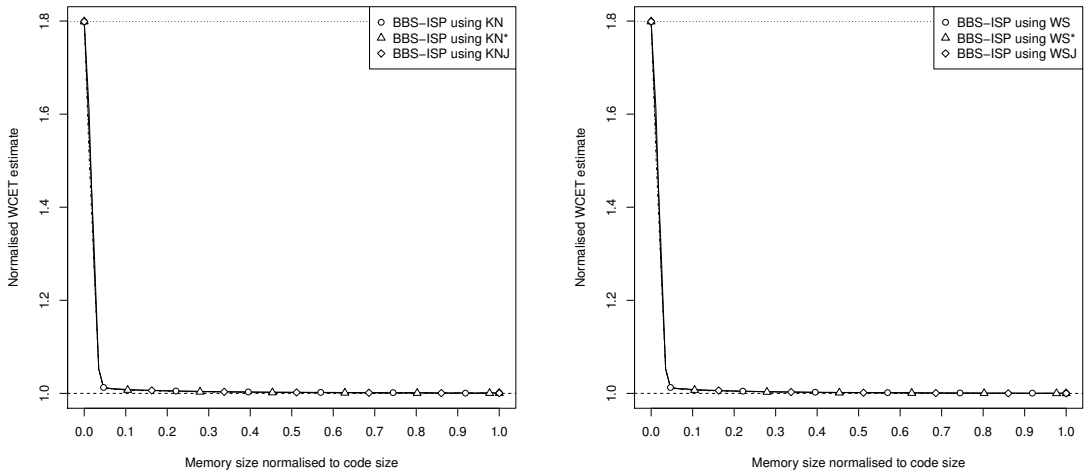
List of Algorithms

Appendix A

S–ISP WCET Evaluations

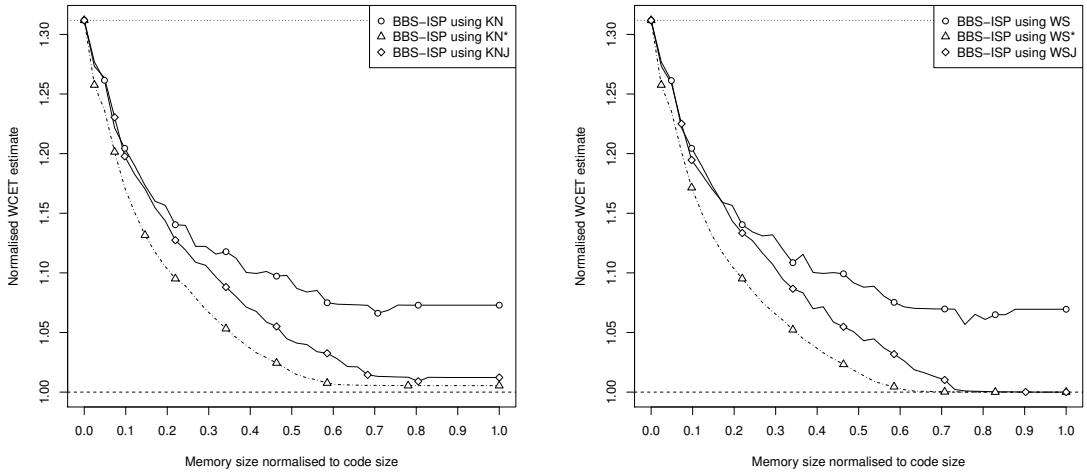
A.1 Comparison of BBS–ISP Assignment Algorithms

In this appendix the figures for WCET estimates of the BBS–ISP as instruction memory for all benchmarks shown in Table 6.12 are provided. In the following the four different basic block assignment algorithms, which are described in Section 4.1.3, are examined: the knapsack-based basic block assignment without additional penalties (denoted as KN), the knapsack-based basic block assignment including jump and size penalties (KNJ), the WCP-sensitive basic block assignment without additional penalties (WS), and the WCP-sensitive basic block assignment including jump and size penalties (WSJ). In addition to that two configurations of the algorithms that does not take penalties into account are depicted: KN* that uses the KN assignment algorithm, but the cost of the basic blocks is not recalculated after their assignment, and WS* that is similar to the KN* configuration, but it uses the WS assignment algorithm. So for the KN* and WS* configurations the estimated execution time will be affected by the same amount as the KN and WS assignment algorithms assume, because no jump and size penalties will be charged on calculation of the estimate after the basic block assignment. In fact the estimates for KN* and WS* underestimate the WCET, since the jump and size penalties have to be taken into account to estimate the WCET, as done for KN and WS. For a preciser discussion of the different estimates refer to Section 6.2.2. For the KNJ and WSJ basic block assignment algorithms the penalties as shown in the Table 6.11 are used. The configurations of the ISPTAP tool used to obtain the estimates is described in Section 6.2.1. The used maximum memory access times (MMATs) model a shared off-chip memory connection, refer to Table 6.9. In the figures the size of the BBS–ISP is varied from 0 to the code size of the benchmark in steps of 32 B. The minimal WCET that is estimated for the case that the whole code fits the BBS–ISP is highlighted with a *dashed* line in the figures. For the BBS–ISP of size 0, i.e. every fetch is obtained from the off-chip memory, the *dotted* line is depicted as the highest possible WCET estimate. The sizes and WCET estimates are normalised to the size of the benchmark and the WCET estimate for the BBS–ISP of the size of the benchmark code, respectively. Notice that for the case that the whole code fits the BBS–ISP the latencies of a physically separated off-chip memory connection can be assumed for WCET estimation. Since for all analysis runs the same memory access times (MMATs) are used, this is not shown in the figures. The use of the latencies of a separated off-chip memory connection for a BBS–ISP that is as large as the benchmark code will lead to a lower (and tighter) WCET estimate.



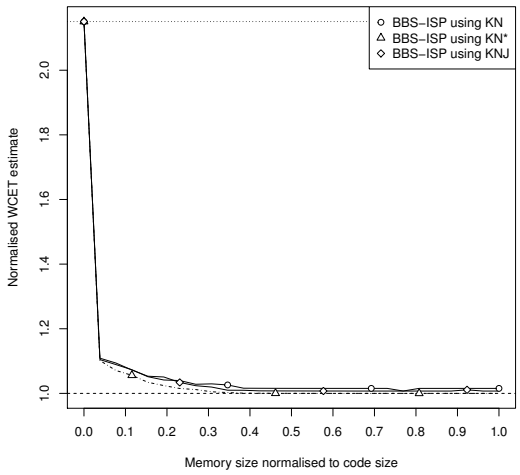
(a) Knapsack-based basic block assignment algorithm (b) WCP-sensitive basic block assignment algorithm

Figure A.1: Normalised WCET estimates for Adpcm using the BBS–ISP with the different basic block assignment algorithms

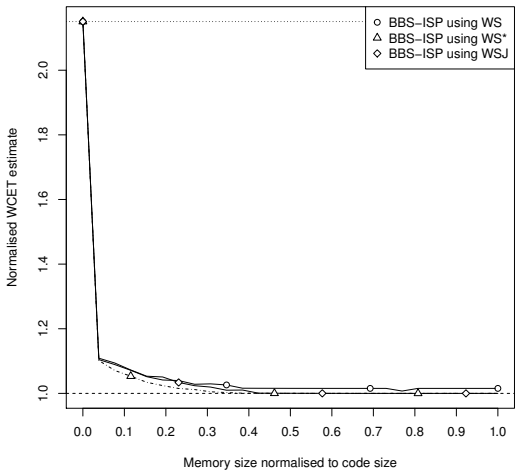


(a) Knapsack-based basic block assignment algorithm (b) WCP-sensitive basic block assignment algorithm

Figure A.2: Normalised WCET estimates for Compress using the BBS–ISP with the different basic block assignment algorithms

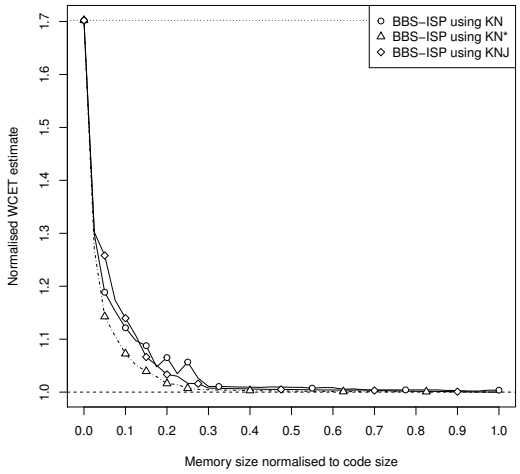


(a) Knapsack-based basic block assignment algorithm

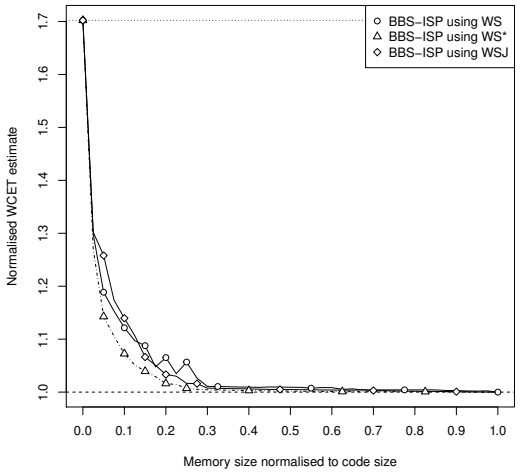


(b) WCP-sensitive basic block assignment algorithm

Figure A.3: Normalised WCET estimates for Dijkstra using the BBS–ISP with the different basic block assignment algorithms

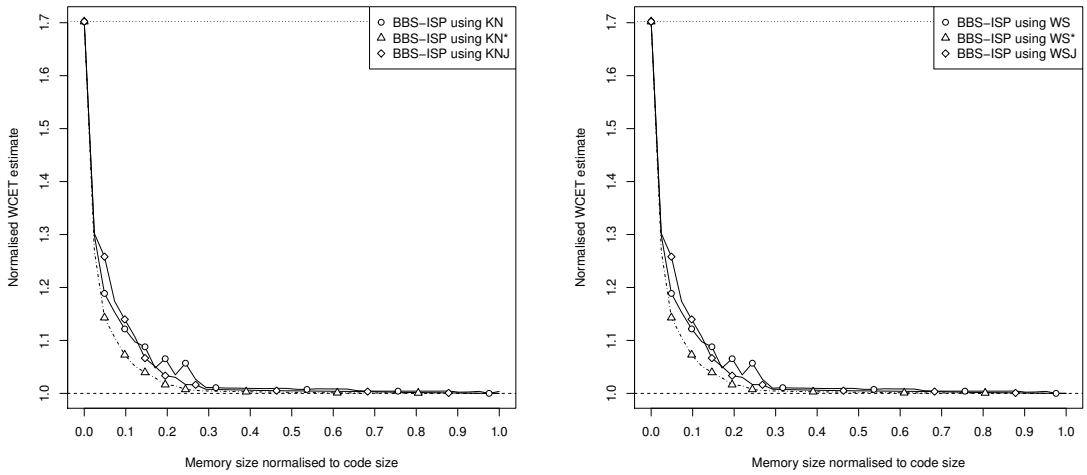


(a) Knapsack-based basic block assignment algorithm



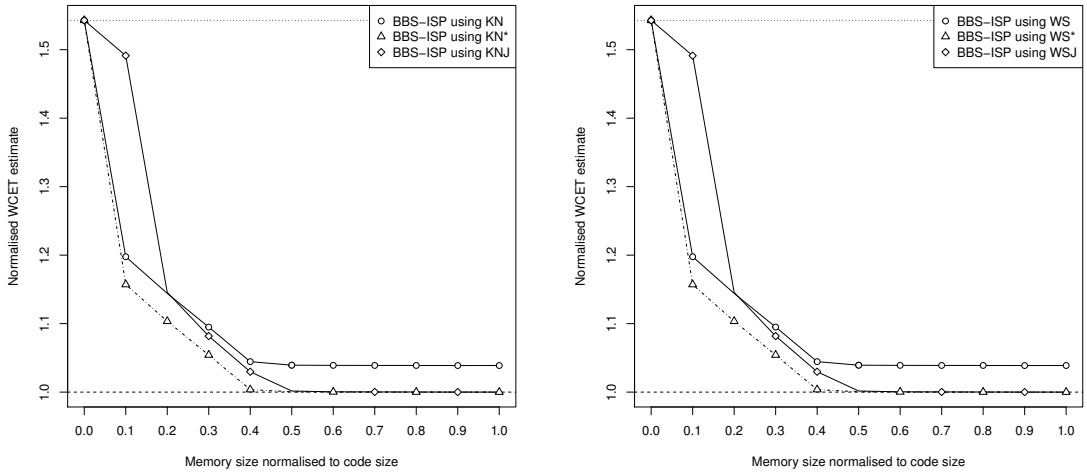
(b) WCP-sensitive basic block assignment algorithm

Figure A.4: Normalised WCET estimates for Edn using the BBS–ISP with the different basic block assignment algorithms



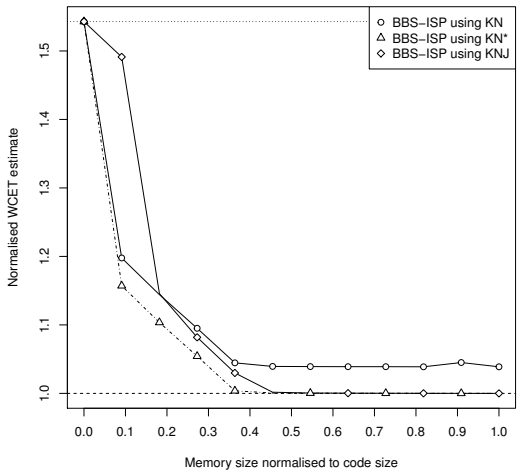
(a) Knapsack-based basic block assignment algorithm (b) WCP-sensitive basic block assignment algorithm

Figure A.5: Normalised WCET estimates for Edn *Loop* using the BBS–ISP with the different basic block assignment algorithms

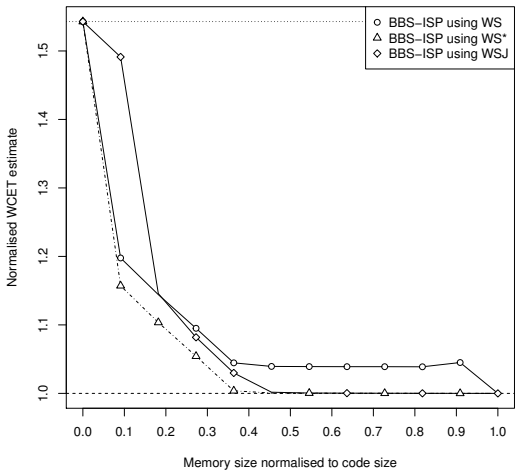


(a) Knapsack-based basic block assignment algorithm (b) WCP-sensitive basic block assignment algorithm

Figure A.6: Normalised WCET estimates for Matmult using the BBS–ISP with the different basic block assignment algorithms

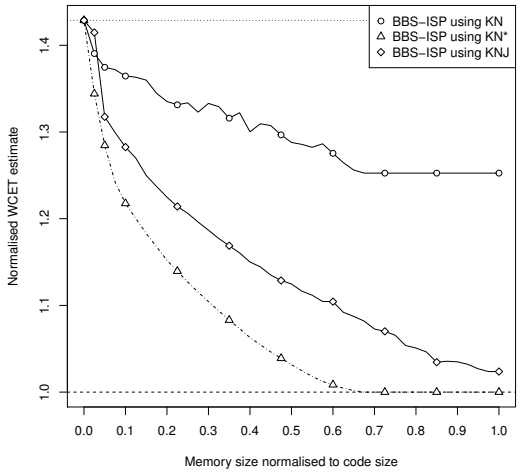


(a) Knapsack-based basic block assignment algorithm

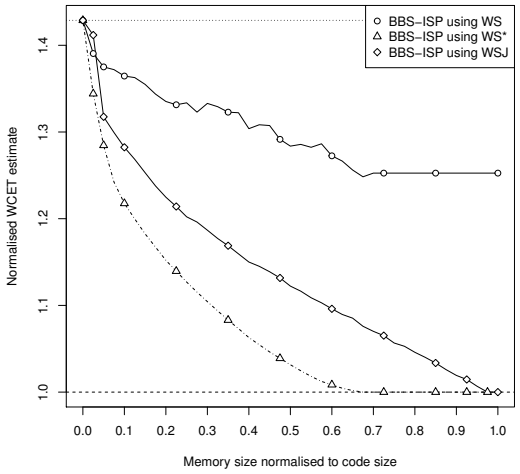


(b) WCP-sensitive basic block assignment algorithm

Figure A.7: Normalised WCET estimates for Matmult *Loop* using the BBS–ISP with the different basic block assignment algorithms

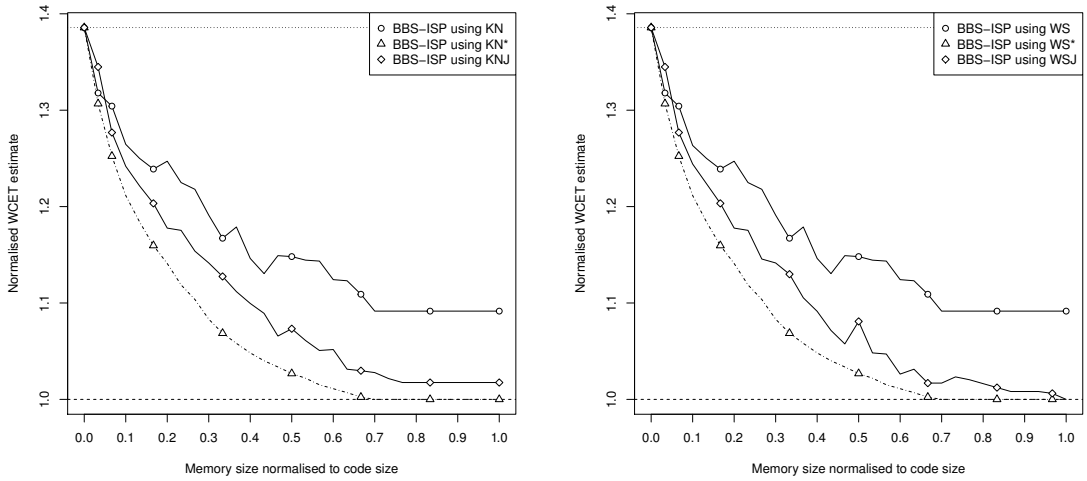


(a) Knapsack-based basic block assignment algorithm



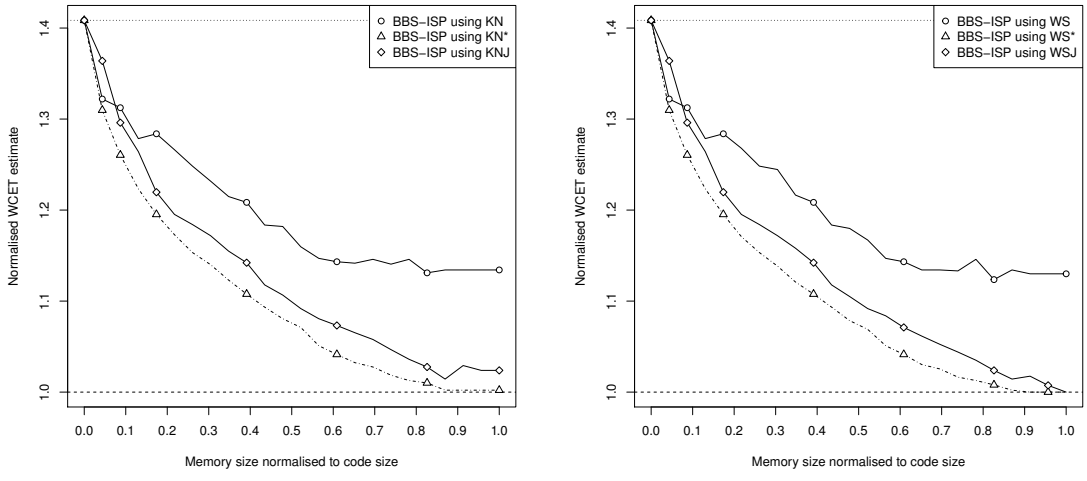
(b) WCP-sensitive basic block assignment algorithm

Figure A.8: Normalised WCET estimates for Puwmod using the BBS–ISP with the different basic block assignment algorithms



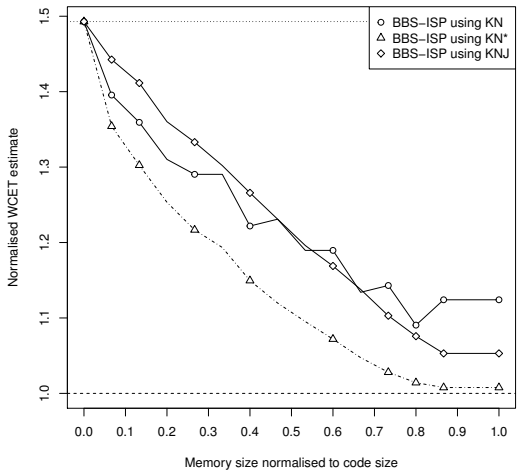
(a) Knapsack-based basic block assignment algorithm (b) WCP-sensitive basic block assignment algorithm

Figure A.9: Normalised WCET estimates for Puwmod *Split* using the BBS–ISP with the different basic block assignment algorithms

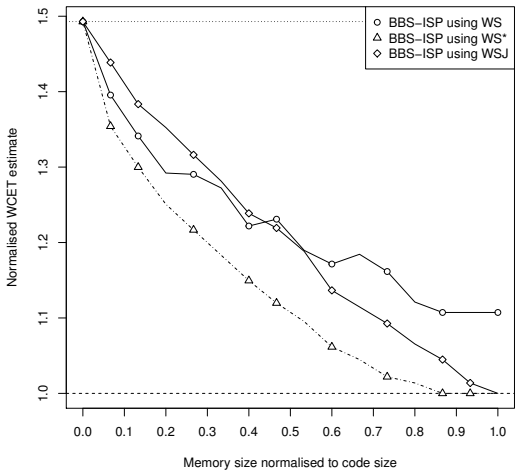


(a) Knapsack-based basic block assignment algorithm (b) WCP-sensitive basic block assignment algorithm

Figure A.10: Normalised WCET estimates for Rspeed using the BBS–ISP with the different basic block assignment algorithms

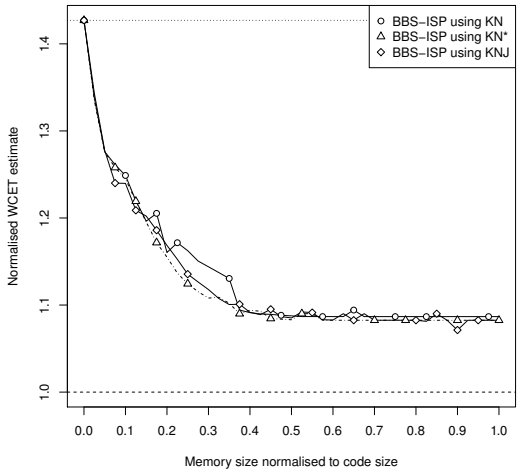


(a) Knapsack-based basic block assignment algorithm

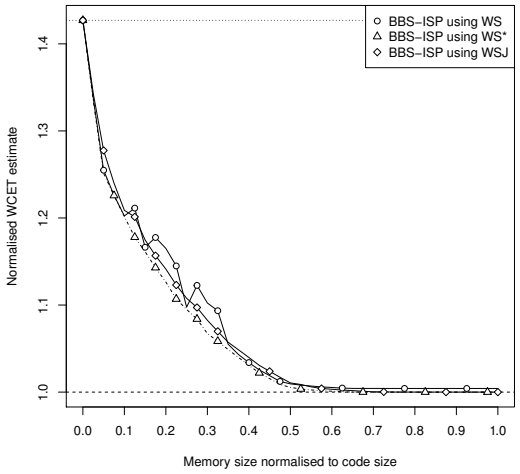


(b) WCP-sensitive basic block assignment algorithm

Figure A.11: Normalised WCET estimates for Rspeed *Split* using the BBS–ISP with the different basic block assignment algorithms

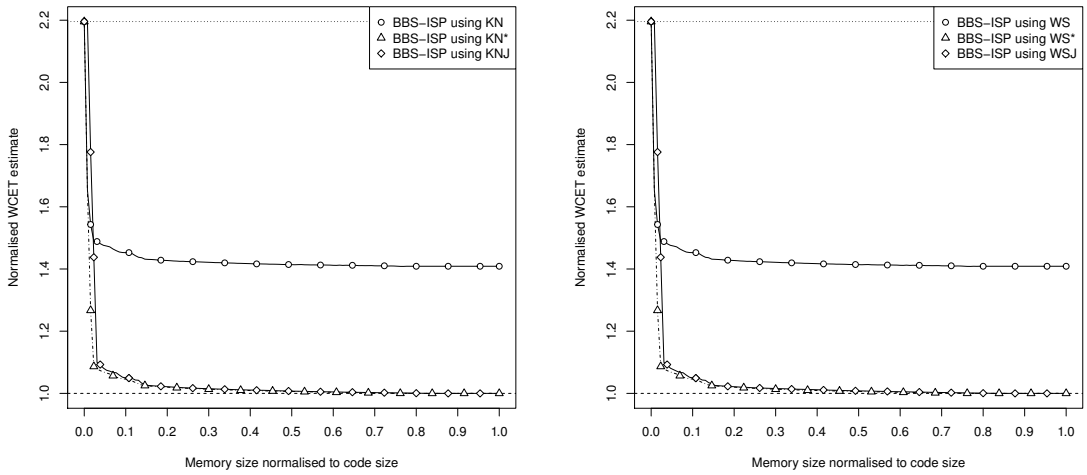


(a) Knapsack-based basic block assignment algorithm



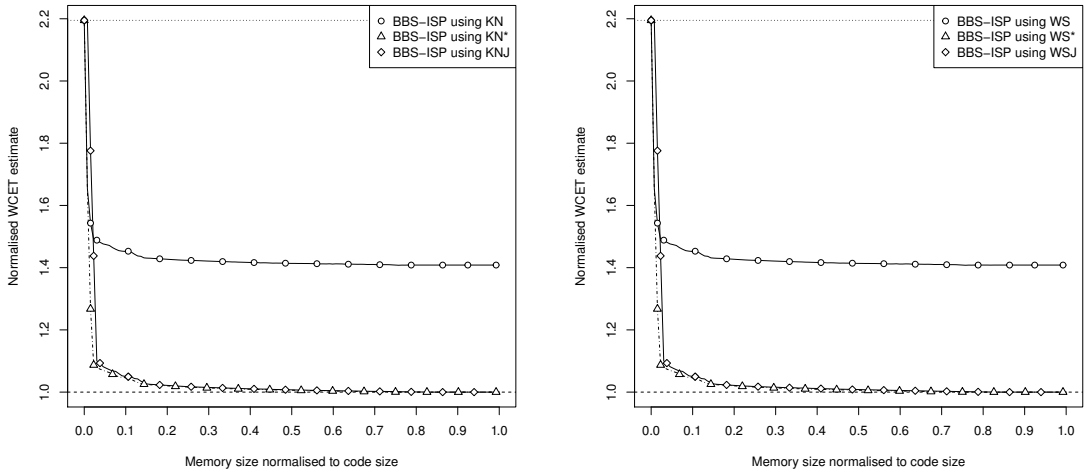
(b) WCP-sensitive basic block assignment algorithm

Figure A.12: Normalised WCET estimates for Sha using the BBS–ISP with the different basic block assignment algorithms



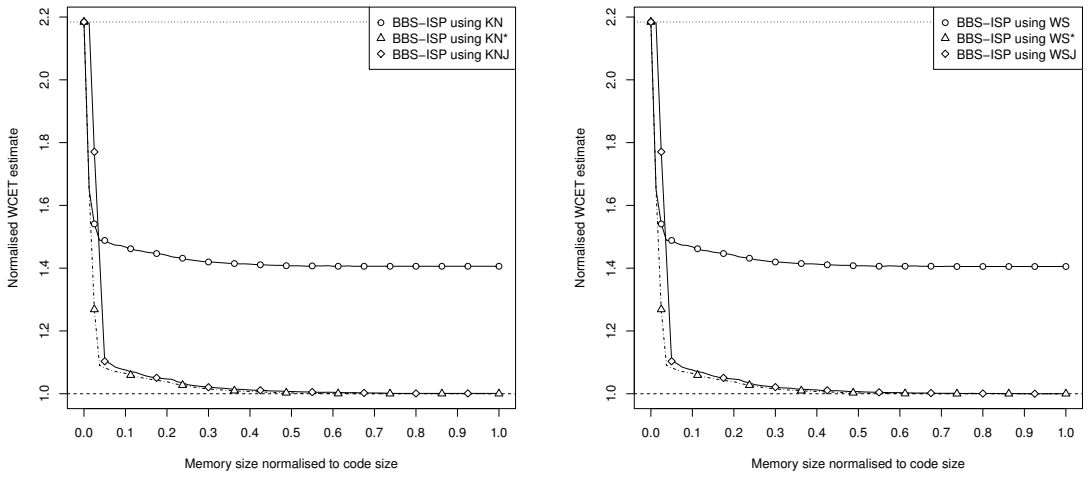
(a) Knapsack-based basic block assignment algorithm (b) WCP-sensitive basic block assignment algorithm

Figure A.13: Normalised WCET estimates for Ttsprk using the BBS–ISP with the different basic block assignment algorithms



(a) Knapsack-based basic block assignment algorithm (b) WCP-sensitive basic block assignment algorithm

Figure A.14: Normalised WCET estimates for Ttsprk *Loop* using the BBS–ISP with the different basic block assignment algorithms



(a) Knapsack-based basic block assignment algorithm (b) WCP-sensitive basic block assignment algorithm

Figure A.15: Normalised WCET estimates for Ttsprk *Split* using the BBS–ISP with the different basic block assignment algorithms

A.2 Comparison of FS–ISP Assignment Algorithms

In this appendix the WCET estimates for the benchmarks listed in Table 6.12 with a FS–ISP as instruction memory are shown. The two function assignment algorithms described in Section 4.1.2 are compared in the following figures: the knapsack-based algorithm (denoted as KN) and the WCP-sensitive algorithm (WS). To depict the progression of the WCET estimates of both assignment algorithms for different FS–ISP sizes, the size of the FS–ISP is varied from 0 to the size of the benchmark in steps of 32 B. The FS–ISP sizes in the Figures A.16 to A.30 are normalised to the code size of the benchmark. Furthermore, the WCET estimates are normalised to the case that the whole benchmark fits the FS–ISP. This optimal WCET estimate is highlighted with a *dashed* line in the figures. The *dotted* line denotes the case that no FS–ISP is used. Both boundary values are numerically the same as the ones in the figures of the evaluation of the different BBS–ISP assignment algorithms in Section A.1. Such that these graphs are comparable. The configurations of the ISPTAP tool used to obtain the estimates is described in Section 6.2.1. For the used memory access times a shared off-chip memory connection is assumed. The number of cycles used for the access times is shown in Table 6.9. Notice that for the case that the whole code fits the FS–ISP the latencies of a physically separated memory connection can be assumed for the WCET estimation, which will provide a lower WCET estimate. Because the same memory latency is used for all analysis runs, this it is not depicted in the figures.

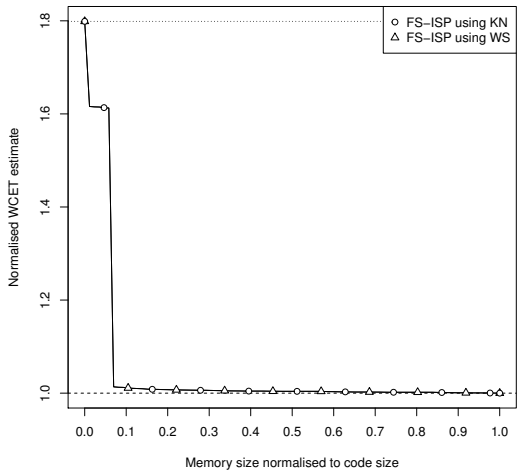


Figure A.16: Normalised WCET estimates for Adpcm using the FS–ISP with the different function assignment algorithms

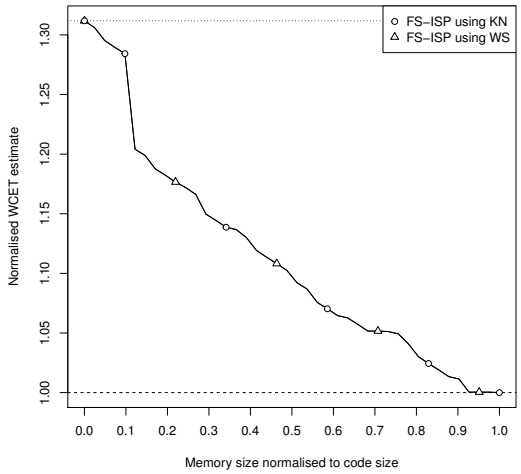


Figure A.17: Normalised WCET estimates for Compress using the FS–ISP with the different function assignment algorithms

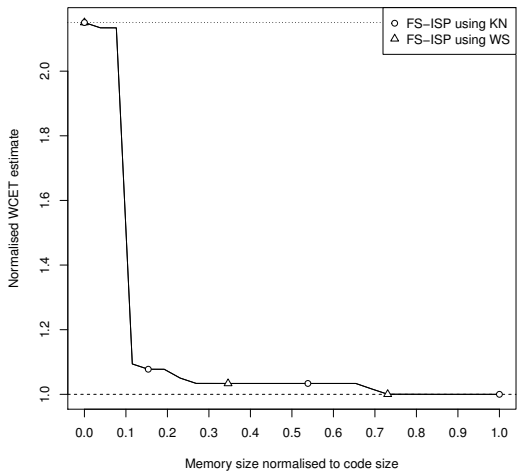


Figure A.18: Normalised WCET estimates for Dijkstra using the FS–ISP with the different function assignment algorithms

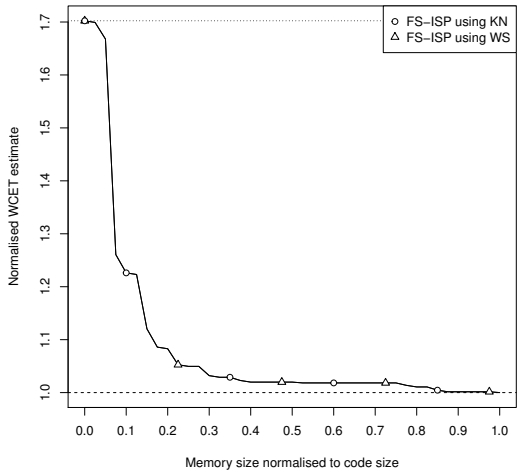


Figure A.19: Normalised WCET estimates for Edn using the FS–ISP with the different function assignment algorithms

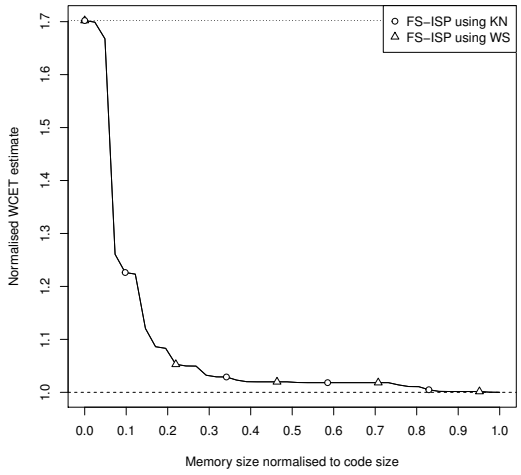


Figure A.20: Normalised WCET estimates for Edn *Loop* using the FS–ISP with the different function assignment algorithms

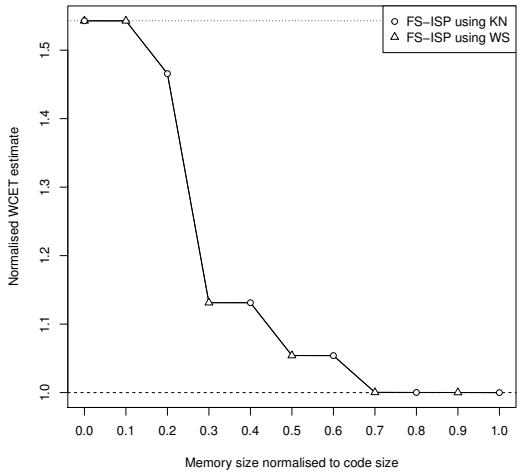


Figure A.21: Normalised WCET estimates for Matmult using the FS–ISP with the different function assignment algorithms

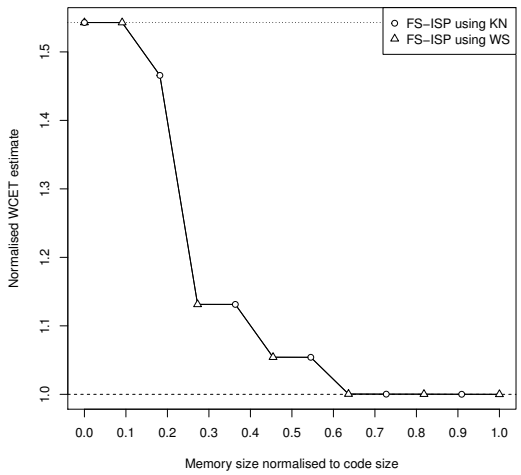


Figure A.22: Normalised WCET estimates for Matmult *Loop* using the FS–ISP with the different function assignment algorithms

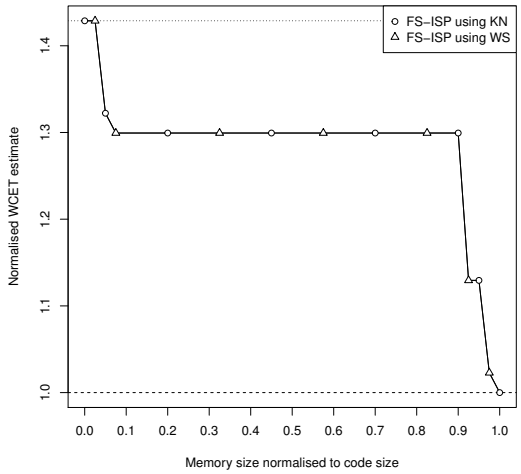


Figure A.23: Normalised WCET estimates for Puwmod using the FS–ISP with the different function assignment algorithms

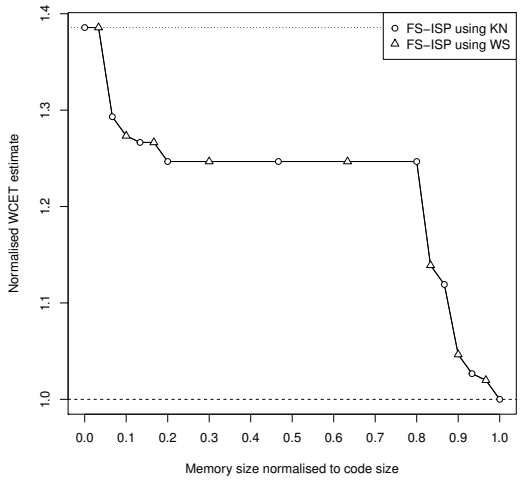


Figure A.24: Normalised WCET estimates for Puwmod *Split* using the FS–ISP with the different function assignment algorithms

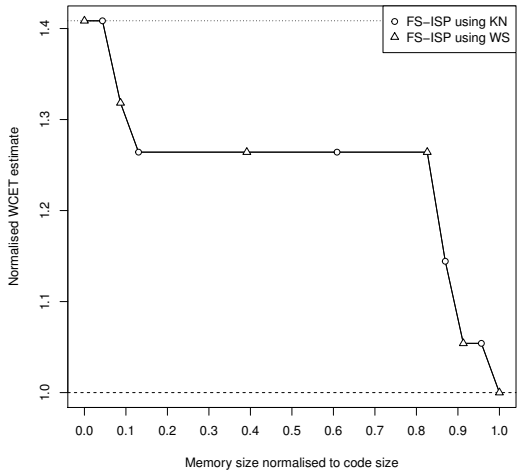


Figure A.25: Normalised WCET estimates for Rspeed using the FS–ISP with the different function assignment algorithms

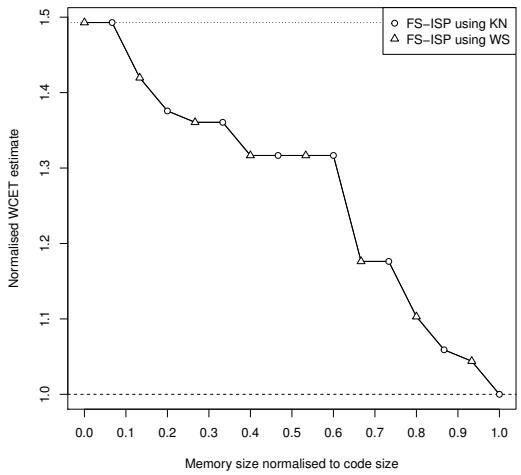


Figure A.26: Normalised WCET estimates for Rspeed *Split* using the FS–ISP with the different function assignment algorithms

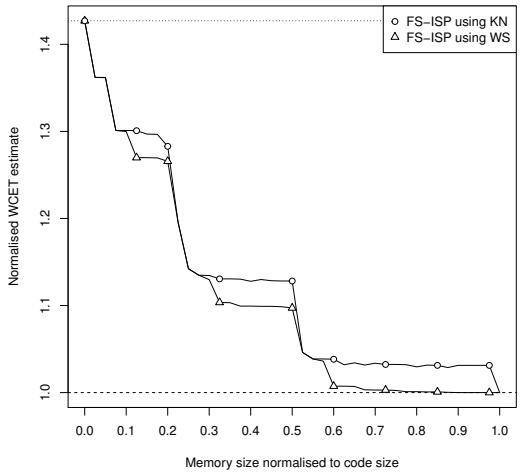


Figure A.27: Normalised WCET estimates for Sha using the FS–ISP with the different function assignment algorithms

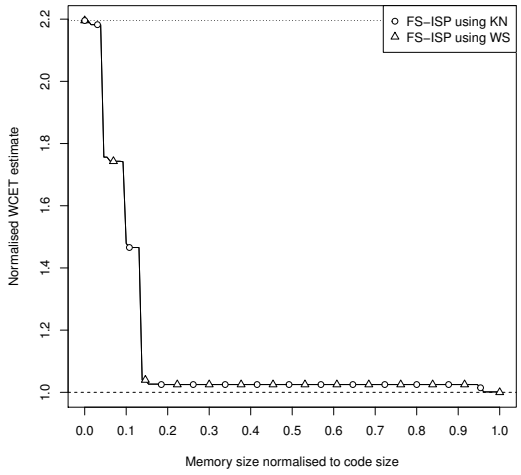


Figure A.28: Normalised WCET estimates for Ttsprk using the FS–ISP with the different function assignment algorithms

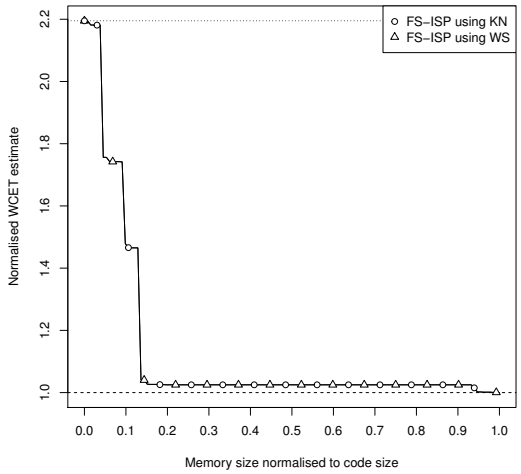


Figure A.29: Normalised WCET estimates for Ttsprk *Loop* using the FS–ISP with the different function assignment algorithms

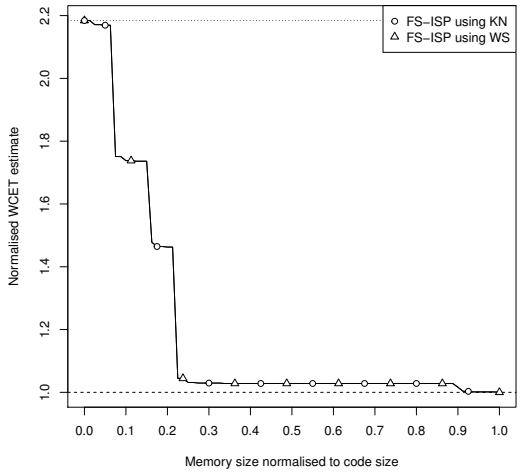


Figure A.30: Normalised WCET estimates for Ttsprk *Split* using the FS–ISP with the different function assignment algorithms

Appendix B

D–ISP WCET Evaluations

B.1 Normalised WCET Estimates

In this appendix normalised WCET estimates for the D–ISP with FIFO replacement policy, static scratchpads (BBS–ISP (WSJ) and FS–ISP (WS)), and caches with different replacement policies (fully associative LRU, fully associative FIFO, and direct mapped) are presented for the benchmarks listed in Table 6.12. The shown data was used for the comparison plots of Section 6.2.3. Also the behaviour of different cache replacement policies is shown, whereas in Section 6.2.3 the D–ISP was compared only to the LRU cache. The configuration of the ISPTAP tool, which was used to obtain the depicted WCET estimates, is described in Section 6.2.1. In all evaluations the interference penalty at the off-chip memory level is assumed for static scratchpads and caches. So the MMATs from the third column of Table 6.9 are applied. Since the D–ISP guarantees the absence of these interferences, the MMATs for an interference free off-chip memory access (as shown in the second column of Table 6.9) are used for the WCET calculation of the D–ISP.

The WCET estimates shown in the Figures B.1 to B.15 are normalised to the minimal possible WCET, which is reached by a system in which all application code fits into a static on-chip memory. Since all code is located in the on-chip memory, no interferences with data accesses at the off-chip memory level can occur. Therefore, for the estimation of the optimal WCET the interference penalty is ignored by using the MMATs for a system with separated off-chip memory connection (refer to Table 6.9). In the figures this optimal WCET is marked by a *dashed* line at a normalised WCET estimate of 1. The *dotted* line represents the usage of no on-chip memory for instructions and so assumes the interference penalty for the memory accesses. It is always the baseline for S–ISP and I–Cache memories for the memory size of 0. The WCET estimates for D–ISP start at the memory size of the largest function in the code, since this is a requirement for the proper use of the D–ISP[1]. The memory size of the different memories compared in the following figures is normalised to the size of the benchmark code (refer to Table 6.12). The memory size of 1 denotes that the whole benchmark code fits the on-chip memory.

For each benchmark two figures are shown: The figures denoted with (a) compare the D–ISP to the BBS–ISP with WSJ assignment algorithm and to the FS–ISP using the WS assignment algorithm. A comparison of the different assignment algorithms for the S–ISP memories that

[1]Smaller D–ISP sizes are possible, but then functions larger than the D–ISP cannot be maintained by the D–ISP and have to be fetched from the off-chip memory. The FMUX (c.f. Section 3.3.2) takes care of routing fetch requests of unmaintained functions to the off-chip memory. But in this case, it cannot be assumed that the off-chip memory connection is free from any interferences.

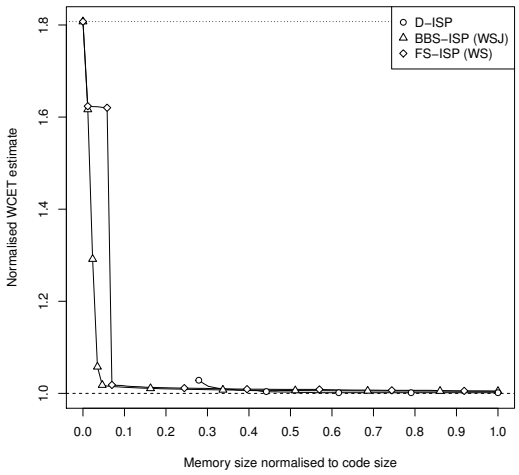
are discussed in Section 4.1 is given in Appendix A. The (b) figures compare the D–ISP with the fully associative LRU cache, the fully associative FIFO cache, and the direct mapped cache. Notice that the WCET estimates of both figure types are normalised to the same value. Thus the WCET estimates of one benchmark can be directly compared for the different memories shown in figures (a) and (b).

For the S–ISP memories with the size of the benchmark the WCET performance is naturally equal to the normalised WCET estimate of 1, which marked by the *dashed* line in the figures. Nevertheless, the figures show for the WCET estimates for a S–ISP of the same size as the benchmark code the estimate with the presence of interferences at the off-chip memory level, whereas the estimate used for normalisation assumes the absence of interferences. This is because for all S–ISP memory sizes these interferences were taken into account during WCET analysis[2]. So to clarify this: the normalised S–ISP WCET estimate at a normalised memory size of 1 is always 1, because off-chip memory interferences can be eliminated. But all values shown in the figures are assuming these interferences, because the same memory latency for all S–ISP sizes is used by the analysis. Because the complete benchmark code is in the on-chip memory a priori, the caches and the D–ISP cannot compete to a S–ISP with a size at least as large as the code. This overhead caused by the dynamic content management, which is discussed in detail in Section 6.2.4.

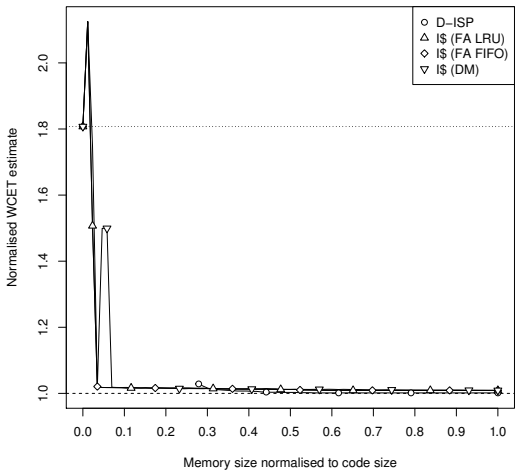
Furthermore, in Table B.1 the baseline WCET estimates in cycles are provided. The second column of the table provides the WCET estimates of a system in which the whole code is located in an on-chip memory and no off-chip memory interferences can occur. These values were used for the normalisation of the WCET estimates in the Figures B.1 to B.15. The third column presents the WCET estimates of the configuration without any on-chip memory. For the description of the benchmarks refer to Table 6.12. The evaluation methodology (e.g. the timing parameters of the memory) that was used to obtain the WCET estimates is described in Section 6.2.1.

The LRU cache, which reaches in most cases the best WCET performance, provides sometimes higher WCET estimates compared to the direct mapped and the FIFO cache as Figure B.2(b), B.3(b), or B.9(b) shows. It is supposed that this effect is caused by the loss of precision that is inherited by the reduction of the state space to a must and may set and the use of abstract transition functions during cache analysis, whereas the direct mapped cache analysis uses different state transition functions and the FIFO cache analysis uses the complete set of reachable states.

[2]Interferences of instruction and data memory accesses at off-chip level aren't possible, if the whole code is located in the on-chip memory. But for smaller memory sizes such interferences can occur and have to be modelled in the timing of the memory access.

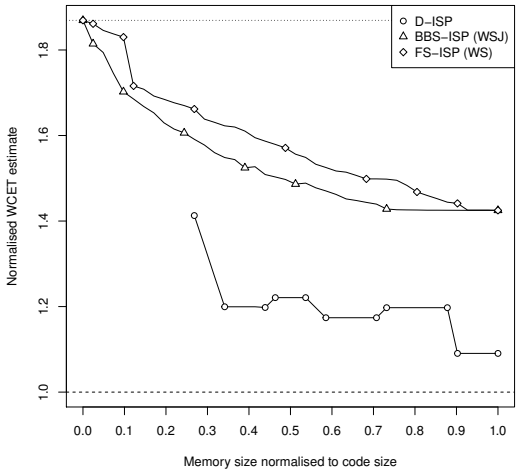


(a) D–ISP, BBS–ISP (WSJ), and FS–ISP (WS)

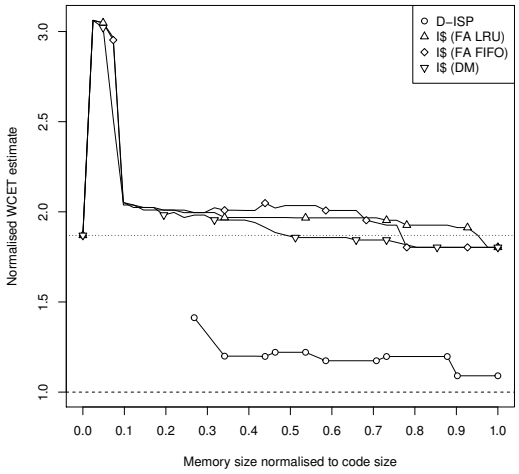


(b) D–ISP and I–Caches (LRU, FIFO, and DM)

Figure B.1: Comparison of the normalised WCET estimates for Adpcm using D–ISP, BBS–ISP, FS–ISP, and I–Cache memories

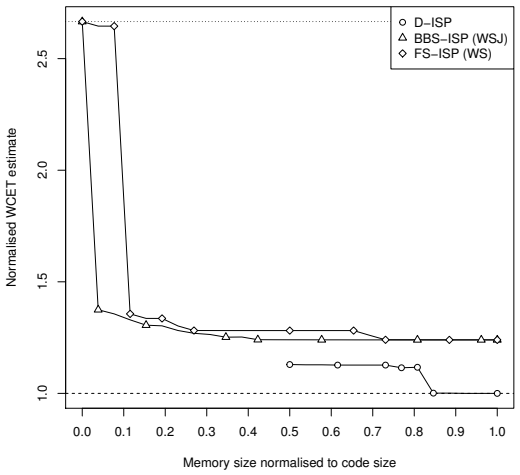


(a) D–ISP, BBS–ISP (WSJ), and FS–ISP (WS)

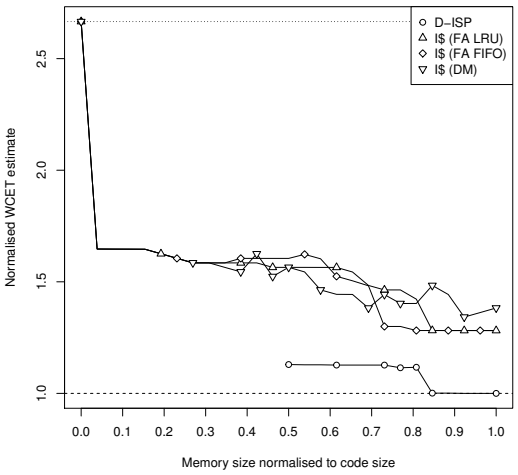


(b) D–ISP and I–Caches (LRU, FIFO, and DM)

Figure B.2: Comparison of the normalised WCET estimates for Compress using D–ISP, BBS–ISP, FS–ISP, and I–Cache memories

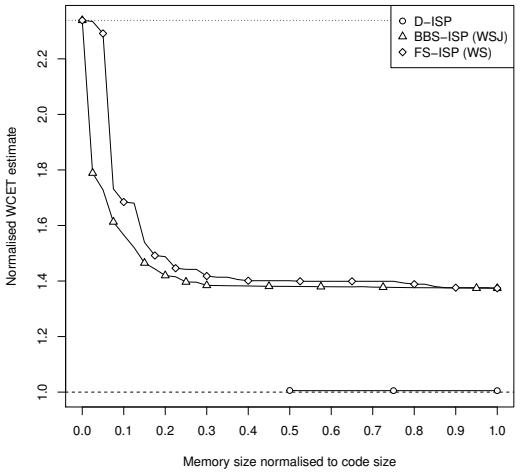


(a) D–ISP, BBS–ISP (WSJ), and FS–ISP (WS)

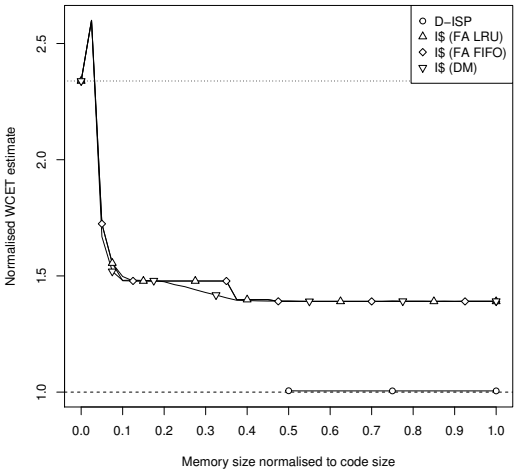


(b) D–ISP and I–Caches (LRU, FIFO, and DM)

Figure B.3: Comparison of the normalised WCET estimates for Dijkstra using D–ISP, BBS–ISP, FS–ISP, and I–Cache memories

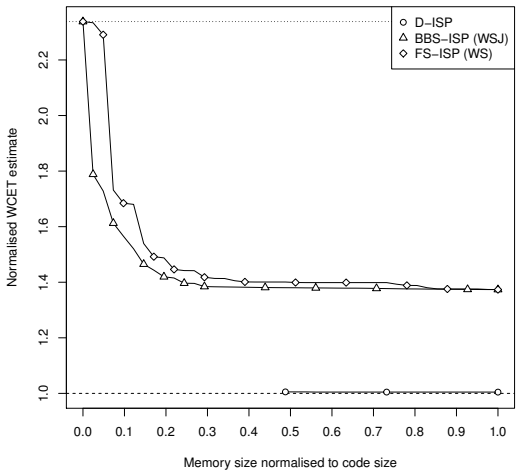


(a) D–ISP, BBS–ISP (WSJ), and FS–ISP (WS)

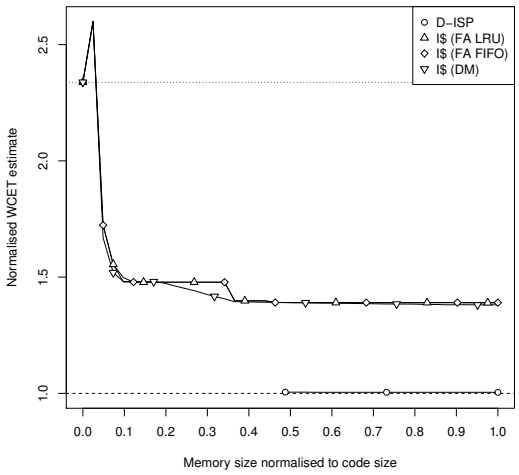


(b) D–ISP and I–Caches (LRU, FIFO, and DM)

Figure B.4: Comparison of the normalised WCET estimates for Edn using D–ISP, BBS–ISP, FS–ISP, and I–Cache memories

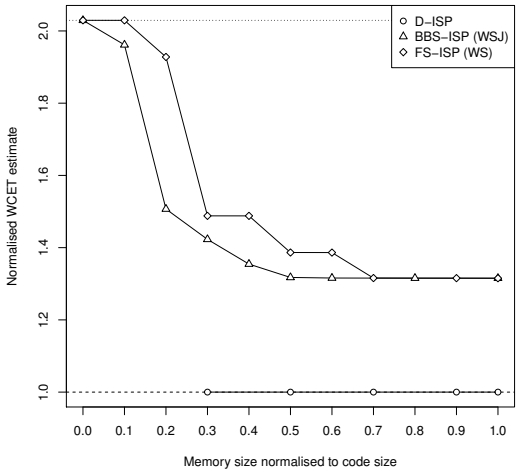


(a) D–ISP, BBS–ISP (WSJ), and FS–ISP (WS)

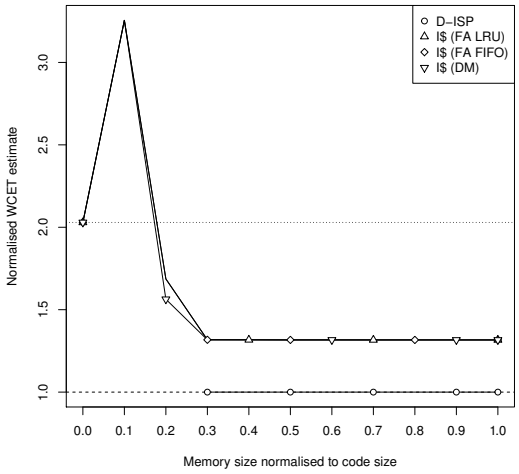


(b) D–ISP and I–Caches (LRU, FIFO, and DM)

Figure B.5: Comparison of the normalised WCET estimates for Edn *Loop* using D–ISP, BBS–ISP, FS–ISP, and I–Cache memories

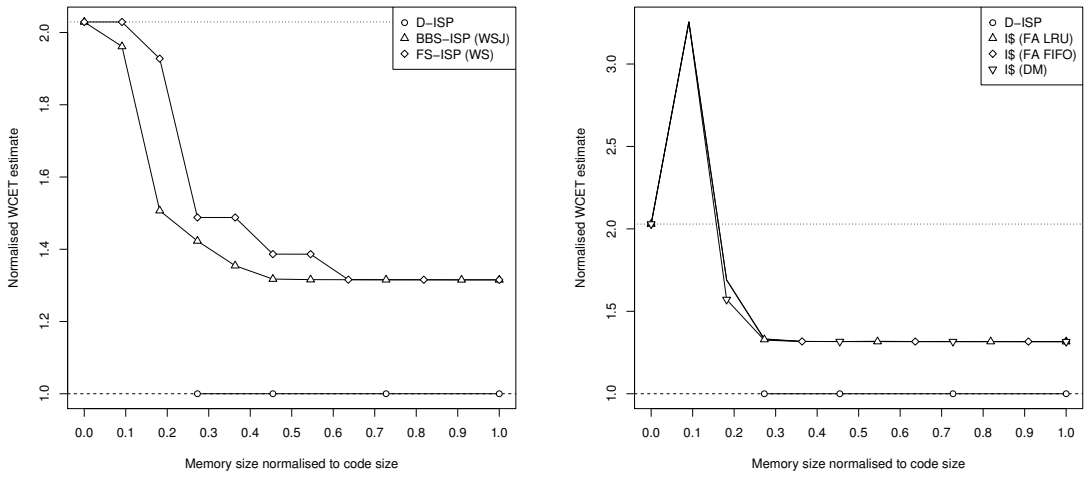


(a) D–ISP, BBS–ISP (WSJ), and FS–ISP (WS)



(b) D–ISP and I–Caches (LRU, FIFO, and DM)

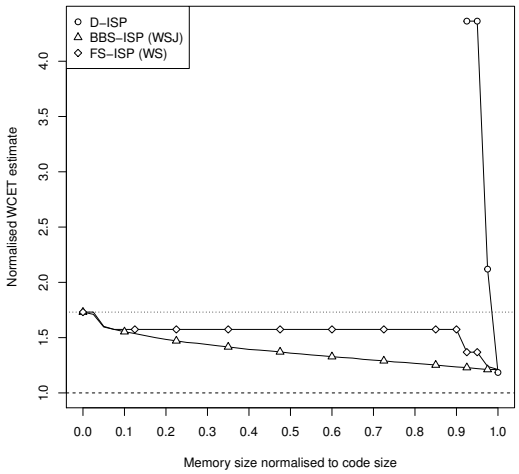
Figure B.6: Comparison of the normalised WCET estimates for Matmult using D–ISP, BBS–ISP, FS–ISP, and I–Cache memories



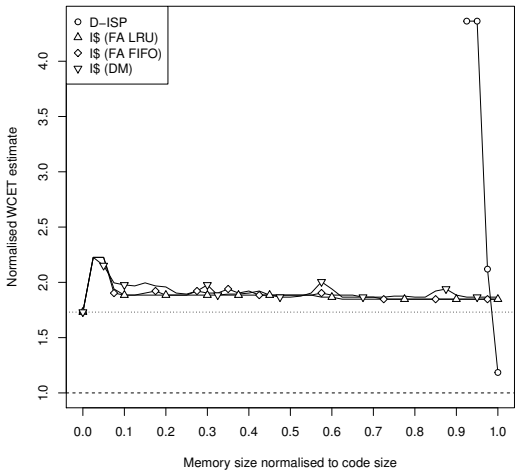
(a) D–ISP, BBS–ISP (WSJ), and FS–ISP (WS)

(b) D–ISP and I–Caches (LRU, FIFO, and DM)

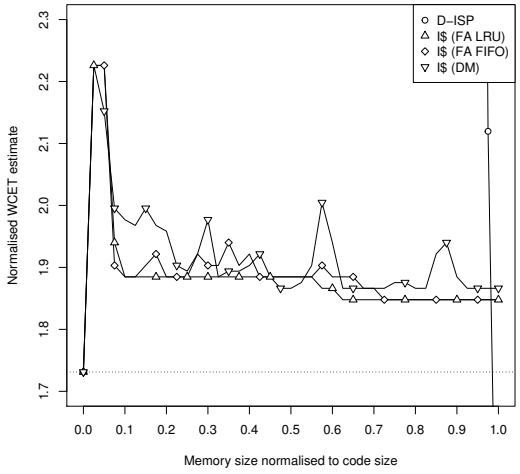
Figure B.7: Comparison of the normalised WCET estimates for Matmult *Loop* using D–ISP, BBS–ISP, FS–ISP, and I–Cache memories



(a) D–ISP, BBS–ISP (WSJ), and FS–ISP (WS)

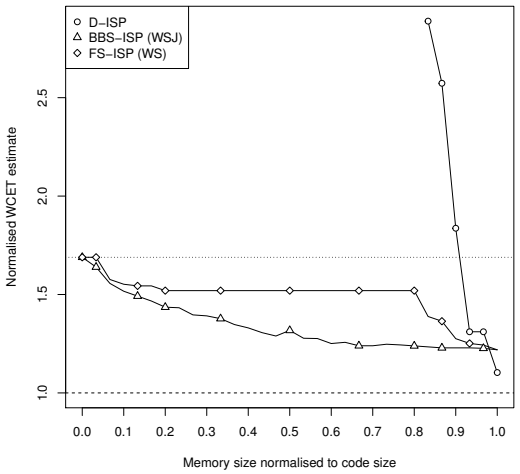


(b) D–ISP and I–Caches (LRU, FIFO, and DM) (see also Figure (c))

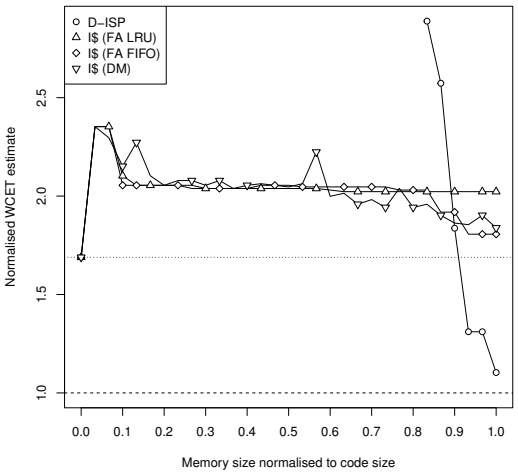


(c) D–ISP and I–Caches (LRU, FIFO, and DM)(zoomed detail of Figure (b))

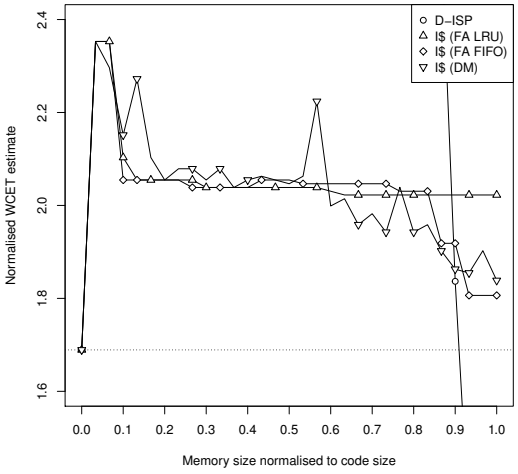
Figure B.8: Comparison of the normalised WCET estimates for Puwmod using D–ISP, BBS–ISP, FS–ISP, and I–Cache memories



(a) D–ISP, BBS–ISP (WSJ), and FS–ISP (WS)

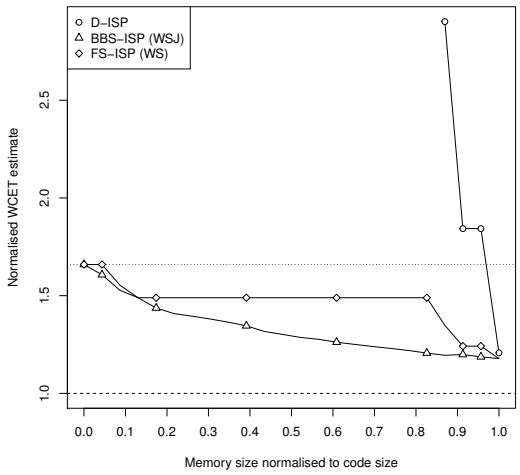


(b) D–ISP and I–Caches (LRU, FIFO, and DM) (see also Figure (c))

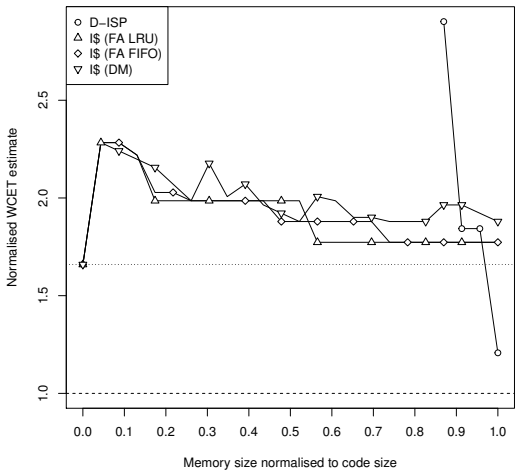


(c) D–ISP and I–Caches (LRU, FIFO, and DM)(zoomed detail of Figure (b))

Figure B.9: Comparison of the normalised WCET estimates for Puwmod *Split* using D–ISP, BBS–ISP, FS–ISP, and I–Cache memories

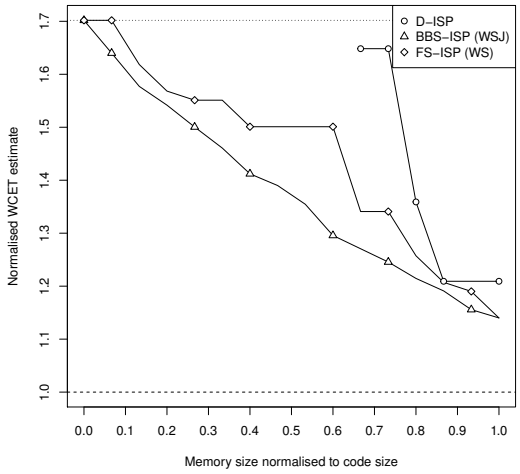


(a) D–ISP, BBS–ISP (WSJ), and FS–ISP (WS)

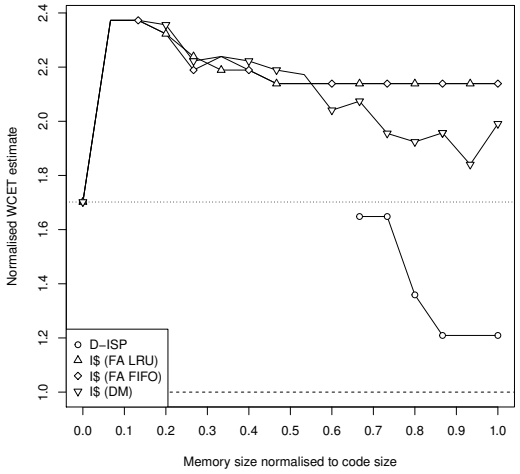


(b) D–ISP and I–Caches (LRU, FIFO, and DM)

Figure B.10: Comparison of the normalised WCET estimates for Rspeed using D–ISP, BBS–ISP, FS–ISP, and I–Cache memories

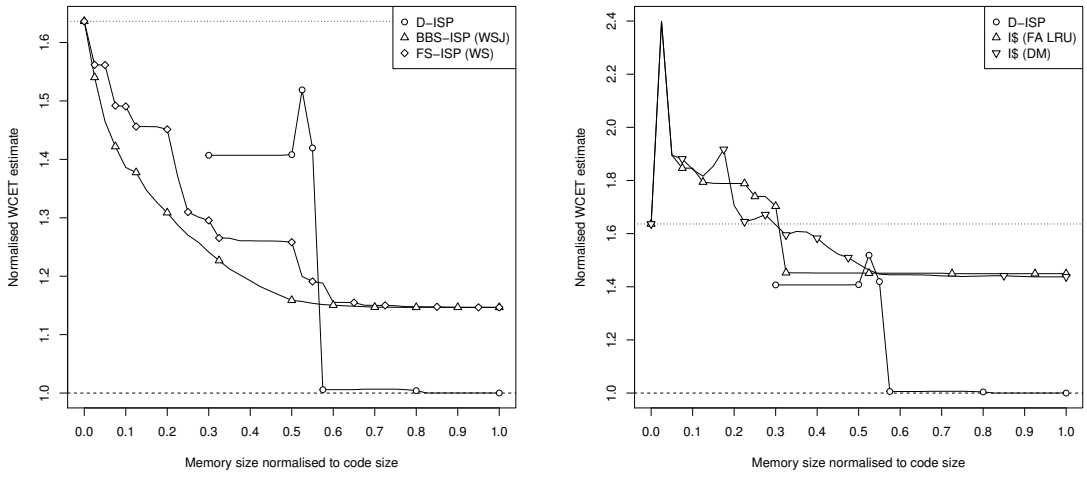


(a) D–ISP, BBS–ISP (WSJ), and FS–ISP (WS)



(b) D–ISP and I–Caches (LRU, FIFO, and DM)

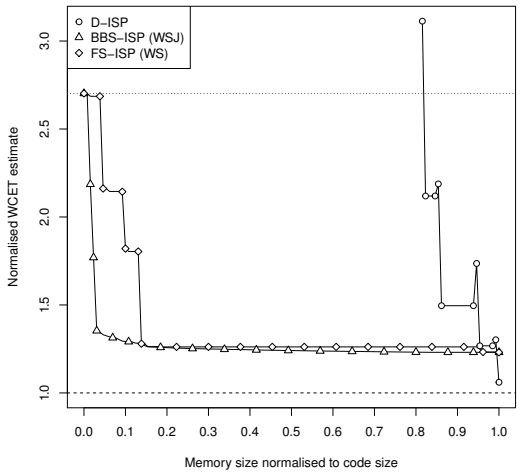
Figure B.11: Comparison of the normalised WCET estimates for Rspeed *Split* using D–ISP, BBS–ISP, FS–ISP, and I–Cache memories



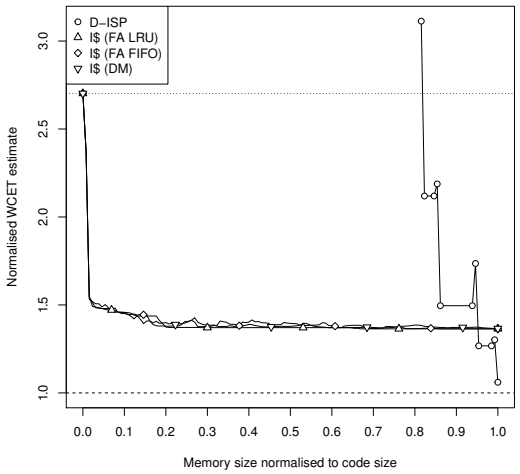
(a) D–ISP, BBS–ISP (WSJ), and FS–ISP (WS)

(b) D–ISP and I–Caches (LRU and DM)

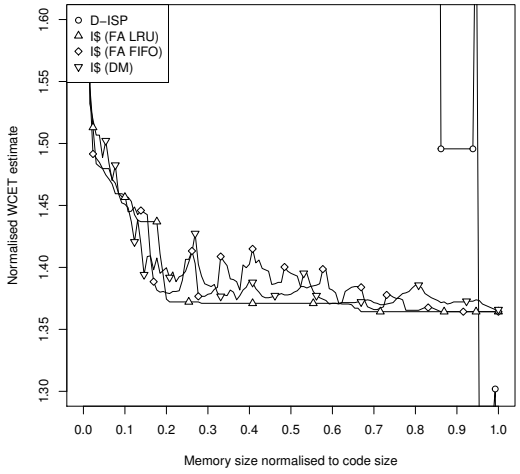
Figure B.12: Comparison of the normalised WCET estimates for Sha using D–ISP, BBS–ISP, FS–ISP, and I–Cache memories (The FIFO cache is not shown, because of too high memory demands during analysis, refer to Section 6.4.2.)



(a) D–ISP, BBS–ISP (WSJ), and FS–ISP (WS)

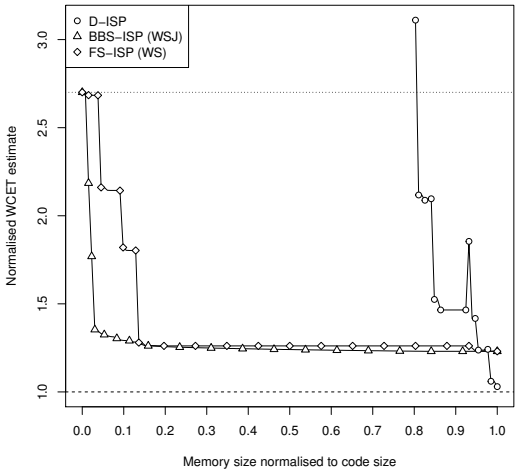


(b) D–ISP and I–Caches (LRU, FIFO, and DM)(see also Figure (c))

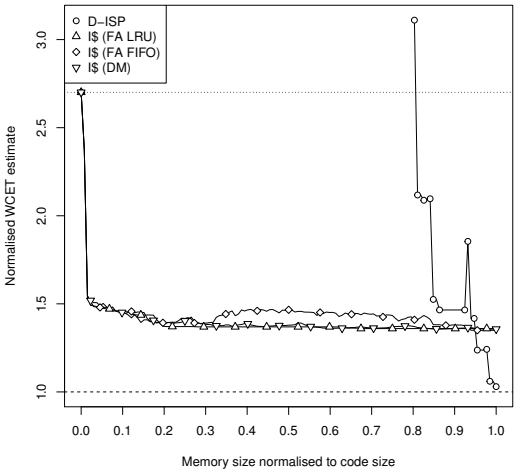


(c) D–ISP and I–Caches (LRU, FIFO, and DM)(zoomed detail of Figure (b))

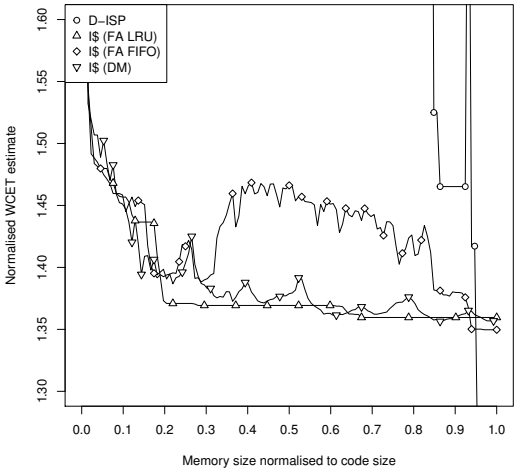
Figure B.13: Comparison of the normalised WCET estimates for Ttsprk using D–ISP, BBS–ISP, FS–ISP, and I–Cache memories



(a) D–ISP, BBS–ISP (WSJ), and FS–ISP (WS)

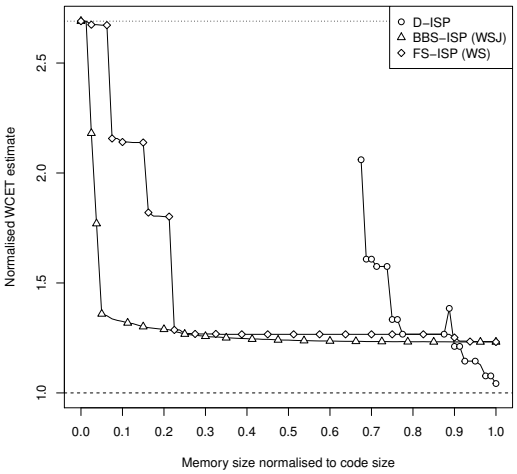


(b) D–ISP and I–Caches (LRU, FIFO, and DM)(see also Figure (c))

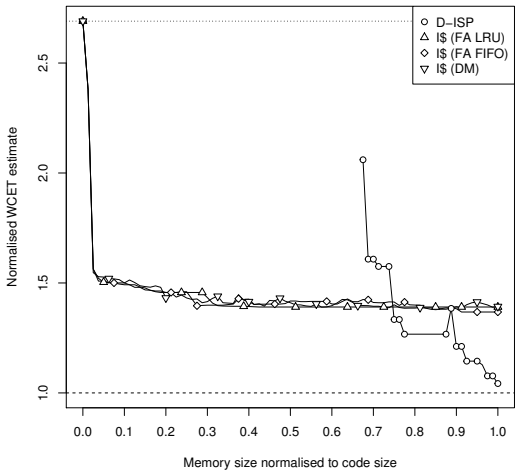


(c) D–ISP and I–Caches (LRU, FIFO, and DM)(zoomed detail of Figure (b))

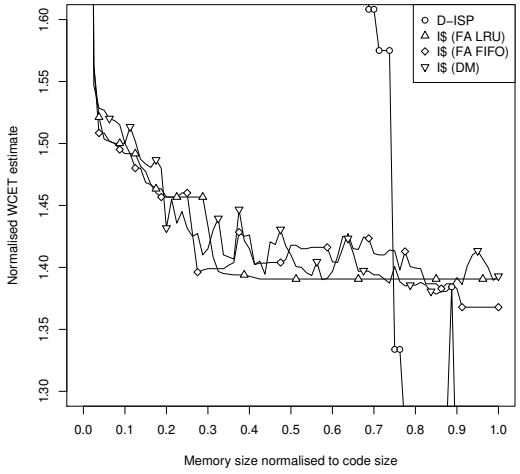
Figure B.14: Comparison of the normalised WCET estimates for Ttsprk *Loop* using D–ISP, BBS–ISP, FS–ISP, and I–Cache memories



(a) D–ISP, BBS–ISP (WSJ), and FS–ISP (WS)



(b) D–ISP and I–Caches (LRU, FIFO, and DM)(see also Figure (c))



(c) D–ISP and I–Caches (LRU, FIFO, and DM)(zoomed detail of Figure (b))

Figure B.15: Comparison of the normalised WCET estimates for Ttsprk *Split* using D–ISP, BBS–ISP, FS–ISP, and I–Cache memories

Table B.1: Estimated WCET baselines for all benchmarks in cycles using on-chip and off-chip memory as instruction memory (The on-chip memory configuration does not charge the interference penalty, since no interferences at the off-chip memory connection can occur. For the off-chip memory configuration the memory interference penalty has to be considered, see Table 6.9.)

Benchmark	**WCET for on-chip memory (in Cycles)**	**WCET for off-chip memory (in Cycles)**
Adpcm	806,701	1,458,000
Compress	125,607	234,804
Edn	89,110	208,405
Edn *Loop*	713,464	1,668,100
Matmult	331,080	671,899
Matmult *Loop*	2,649,224	5,376,052
Puwmod	3,686	6,381
Puwmod *Split*	4,220	7,128
Rspeed	1,600	2,655
Rspeed *Split*	2,036	3,465
Sha	7,253,270	11,868,855
Ttsprk	60,370	163,133
Ttsprk *Loop*	483,551	1,306,036
Ttsprk *Split*	61,168	164,573

B.2 Impact of the Off-Chip Memory Hierarchy

The Tables B.2 to B.16 provide the raw data for the columns 6 and 7 of Table 6.16 (refer to Section 6.2.5). The difference of the WCET estimate for a SHMC (shared off-chip memory connection) and SEMC (separated off-chip memory connection) system is shown for the LRU cache. The SHMC WCET estimates assume that every off-chip memory access interferes, whereas in the SEMC estimates no interferences have to be taken into account. The shown difference between both estimates quantifies the range of the impact that is caused by the off-chip memory interferences for an LRU cache.

Furthermore, the difference of the WCET estimates for D–ISP with FIFO and stack-based replacement policy and the SHMC WCET estimates for the LRU cache is shown in the tables. If the differences of D–ISP and the SHMC cache estimates are larger than the difference of the SHMC and SEMC estimates for the cache, the D–ISP outperforms the cache in a system with shared off-chip memory connection, because the D–ISP's performance upper-bound is even below the WCET estimate for a cache in a system in which no interferences can occur. Therefore, the D–ISP has a safe WCET estimate that is below any safe WCET estimate of a cache in a system with shared off-chip memory connection. Otherwise if the values are positive but not larger than the SHMC/SEMC difference for the cache, the D–ISP gives lower WCET guarantees than the analysis of the LRU cache with shared off-chip memory connection. Anyhow, in this case the D–ISP does not necessarily perform better than the cache in a system with shared off-chip memory connection, but tighter WCET estimates are given by the analysis. An integrated analysis that limits the number of interferences and provides more precise WCET estimates than the cache analysis performed by ISPTAP, is necessary to determine which memory actually has a better WCET performance in such cases.

The memory sizes in the tables are normalised to the size of the benchmark (see Table 6.12). For memory sizes smaller than the largest function no values for the D–ISP are shown.

Table B.2: Impact of the interference penalty on the WCET estimates for Adpcm (The WCET differences are relative to the WCET estimate of the LRU cache with interference penalty.)

Norm. Memory Size	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**	**Norm. Memory Size**	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**
0.01	25.03%	-	-	0.51	0.79%	0.87%	1.00%
0.02	16.00%	-	-	0.52	0.79%	0.93%	0.99%
0.03	1.11%	-	-	0.53	0.75%	0.86%	0.91%
0.05	1.07%	-	-	0.55	0.73%	0.82%	0.87%
0.06	1.07%	-	-	0.56	0.72%	0.85%	0.84%
0.07	1.07%	-	-	0.57	0.69%	0.79%	0.78%
0.08	1.05%	-	-	0.58	0.67%	0.74%	0.74%
0.09	1.04%	-	-	0.59	0.67%	0.74%	0.74%
0.10	1.01%	-	-	0.60	0.67%	0.74%	0.74%
0.12	1.00%	-	-	0.62	0.67%	0.74%	0.74%
0.13	1.00%	-	-	0.63	0.67%	0.74%	0.74%
0.14	1.00%	-	-	0.64	0.67%	0.74%	0.74%
0.15	1.00%	-	-	0.65	0.67%	0.74%	0.74%
0.16	1.00%	-	-	0.66	0.67%	0.74%	0.74%
0.17	1.00%	-	-	0.67	0.67%	0.74%	0.74%
0.19	1.00%	-	-	0.69	0.67%	0.74%	0.74%
0.20	0.99%	-	-	0.70	0.67%	0.74%	0.74%
0.21	0.98%	-	-	0.71	0.67%	0.74%	0.74%
0.22	0.98%	-	-	0.72	0.67%	0.74%	0.74%
0.23	0.98%	-	-	0.73	0.67%	0.74%	0.74%
0.24	0.98%	-	-	0.74	0.67%	0.74%	0.74%
0.26	0.98%	-	-	0.76	0.67%	0.74%	0.74%
0.27	0.98%	-	-	0.77	0.67%	0.74%	0.74%
0.28	0.96%	-1.30%	-0.90%	0.78	0.67%	0.74%	0.74%
0.29	0.95%	-0.63%	0.28%	0.79	0.67%	0.74%	0.74%
0.30	0.93%	-0.05%	0.75%	0.80	0.67%	0.74%	0.74%
0.31	0.92%	0.12%	1.13%	0.81	0.67%	0.74%	0.74%
0.33	0.90%	0.29%	1.10%	0.83	0.67%	0.74%	0.74%
0.34	0.88%	0.35%	1.05%	0.84	0.67%	0.74%	0.74%
0.35	0.87%	0.43%	1.13%	0.85	0.67%	0.74%	0.74%
0.36	0.86%	0.46%	1.11%	0.86	0.67%	0.74%	0.74%
0.37	0.86%	0.60%	1.10%	0.87	0.67%	0.74%	0.74%
0.38	0.86%	0.60%	1.12%	0.88	0.67%	0.74%	0.74%
0.40	0.86%	0.60%	1.14%	0.90	0.67%	0.74%	0.74%
0.41	0.85%	0.68%	1.13%	0.91	0.67%	0.74%	0.74%
0.42	0.83%	0.64%	1.09%	0.92	0.67%	0.74%	0.74%
0.43	0.82%	0.78%	1.06%	0.93	0.67%	0.74%	0.74%
0.44	0.81%	0.80%	1.03%	0.94	0.67%	0.74%	0.74%
0.45	0.81%	0.85%	1.03%	0.95	0.67%	0.74%	0.74%
0.47	0.81%	0.85%	1.03%	0.97	0.67%	0.74%	0.74%
0.48	0.80%	0.89%	1.02%	0.98	0.67%	0.74%	0.74%
0.49	0.80%	0.89%	1.02%	0.99	0.67%	0.74%	0.74%
0.50	0.80%	0.89%	1.02%	1.00	0.67%	0.74%	0.74%
				Average	**1.26%**	**0.66%**	**0.81%**

Table B.3: Impact of the interference penalty on the WCET estimates for Compress (The WCET differences are relative to the WCET estimate of the LRU cache with interference penalty.)

Norm. Memory Size	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**
0.02	39.04%	-	-
0.05	39.00%	-	-
0.07	38.78%	-	-
0.10	35.09%	-	-
0.12	35.01%	-	-
0.15	34.84%	-	-
0.17	34.84%	-	-
0.20	34.84%	-	-
0.22	34.84%	-	-
0.24	34.84%	-	-
0.27	34.75%	29.21%	30.41%
0.29	34.75%	32.75%	32.75%
0.32	34.75%	36.36%	39.90%
0.34	34.58%	39.06%	39.06%
0.37	34.58%	39.06%	39.06%
0.39	34.58%	39.06%	39.13%
0.41	34.58%	39.06%	39.13%
0.44	34.58%	39.14%	39.13%
0.46	34.58%	37.97%	40.35%
0.49	34.58%	37.97%	40.35%
0.51	34.57%	37.92%	40.30%
0.54	34.57%	37.92%	40.30%
0.56	34.57%	39.11%	40.30%
0.59	34.57%	40.31%	40.30%
0.61	34.57%	40.31%	40.30%
0.63	34.57%	40.31%	40.30%
0.66	34.57%	40.31%	40.30%
0.68	34.57%	40.31%	40.30%
0.71	34.57%	40.31%	40.30%
0.73	34.49%	38.69%	43.50%
0.76	34.49%	38.69%	43.50%
0.78	34.31%	37.83%	42.71%
0.80	34.31%	37.83%	42.71%
0.83	34.31%	37.83%	42.71%
0.85	34.31%	37.83%	42.71%
0.88	34.31%	37.83%	42.71%
0.90	34.22%	42.98%	42.30%
0.93	34.22%	42.98%	42.30%
0.95	33.90%	41.71%	41.01%
0.98	33.40%	39.52%	38.80%
1.00	33.40%	39.52%	38.80%
Average	**34.84%**	**38.70%**	**40.18%**

Table B.4: Impact of the interference penalty on the WCET estimates for Dijkstra (The WCET differences are relative to the WCET estimate of the LRU cache with interference penalty.)

Norm. Memory Size	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**
0.04	26.19%	-	-
0.08	26.19%	-	-
0.12	26.18%	-	-
0.15	26.18%	-	-
0.19	25.92%	-	-
0.23	25.65%	-	-
0.27	25.37%	-	-
0.31	25.37%	-	-
0.35	25.37%	-	-
0.38	25.37%	-	-
0.42	25.37%	-	-
0.46	25.09%	-	-
0.50	25.09%	27.83%	27.83%
0.54	25.09%	27.90%	27.90%
0.58	25.09%	27.89%	27.97%
0.62	25.09%	27.97%	27.97%
0.65	24.80%	27.02%	27.02%
0.69	23.90%	24.05%	24.05%
0.73	23.58%	23.01%	31.39%
0.77	23.58%	23.83%	31.39%
0.81	22.89%	21.46%	29.39%
0.85	20.25%	21.90%	21.66%
0.88	20.24%	21.88%	21.64%
0.92	20.24%	21.96%	21.64%
0.96	20.24%	21.96%	21.64%
1.00	20.24%	21.96%	21.64%
Average	**24.18%**	**24.33%**	**25.94%**

Table B.5: Impact of the interference penalty on the WCET estimates for Edn (The WCET differences are relative to the WCET estimate of the LRU cache with interference penalty.)

Norm. Memory Size	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**
0.03	36.57%	-	-
0.05	31.25%	-	-
0.08	29.53%	-	-
0.10	28.63%	-	-
0.13	28.62%	-	-
0.15	28.61%	-	-
0.18	28.61%	-	-
0.20	28.61%	-	-
0.23	28.61%	-	-
0.25	28.61%	-	-
0.28	28.61%	-	-
0.30	28.61%	-	-
0.33	28.61%	-	-
0.35	28.61%	-	-
0.38	27.56%	-	-
0.40	27.56%	-	-
0.43	27.56%	-	-
0.45	27.56%	-	-
0.48	27.45%	-	-
0.50	27.45%	27.67%	27.72%
0.53	27.45%	27.70%	27.70%
0.55	27.45%	27.70%	27.70%
0.58	27.45%	27.70%	27.70%
0.60	27.45%	27.70%	27.70%
0.63	27.45%	27.70%	27.75%
0.65	27.45%	27.70%	27.75%
0.68	27.45%	27.70%	27.75%
0.70	27.45%	27.70%	27.75%
0.73	27.45%	27.70%	27.75%
0.75	27.45%	27.70%	27.75%
0.78	27.45%	27.70%	27.75%
0.80	27.45%	27.70%	27.75%
0.83	27.45%	27.70%	27.75%
0.85	27.45%	27.70%	27.75%
0.88	27.45%	27.70%	27.75%
0.90	27.45%	27.70%	27.75%
0.93	27.45%	27.70%	27.75%
0.95	27.45%	27.70%	27.75%
0.98	27.45%	27.70%	27.75%
1.00	27.45%	27.70%	27.75%
Average	**28.16%**	**27.70%**	**27.74%**

Table B.6: Impact of the interference penalty on the WCET estimates for Edn *Loop* (The WCET differences are relative to the WCET estimate of the LRU cache with interference penalty.)

Norm. Memory Size	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**
0.02	36.56%	-	-
0.05	31.23%	-	-
0.07	29.52%	-	-
0.10	28.62%	-	-
0.12	28.61%	-	-
0.15	28.60%	-	-
0.17	28.60%	-	-
0.20	28.60%	-	-
0.22	28.60%	-	-
0.24	28.60%	-	-
0.27	28.60%	-	-
0.29	28.60%	-	-
0.32	28.60%	-	-
0.34	28.60%	-	-
0.37	27.55%	-	-
0.39	27.55%	-	-
0.41	27.55%	-	-
0.44	27.55%	-	-
0.46	27.44%	-	-
0.49	27.44%	27.72%	27.76%
0.51	27.44%	27.70%	27.74%
0.54	27.44%	27.70%	27.74%
0.56	27.44%	27.74%	27.74%
0.59	27.44%	27.74%	27.74%
0.61	27.43%	27.73%	27.77%
0.63	27.43%	27.73%	27.77%
0.66	27.43%	27.73%	27.77%
0.68	27.43%	27.73%	27.77%
0.71	27.43%	27.73%	27.77%
0.73	27.43%	27.73%	27.77%
0.76	27.43%	27.73%	27.77%
0.78	27.43%	27.73%	27.77%
0.80	27.43%	27.73%	27.77%
0.83	27.43%	27.73%	27.77%
0.85	27.43%	27.73%	27.77%
0.88	27.43%	27.73%	27.77%
0.90	27.43%	27.73%	27.77%
0.93	27.42%	27.69%	27.74%
0.95	27.42%	27.69%	27.74%
0.98	27.42%	27.69%	27.74%
1.00	27.42%	27.70%	27.74%
Average	**28.13%**	**27.72%**	**27.76%**

Table B.7: Impact of the interference penalty on the WCET estimates for Matmult (The WCET differences are relative to the WCET estimate of the LRU cache with interference penalty.)

Norm. Memory Size	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**
0.10	37.73%	-	-
0.20	29.06%	-	-
0.30	24.01%	24.09%	24.09%
0.40	24.00%	24.07%	24.07%
0.50	24.00%	24.04%	24.04%
0.60	24.00%	24.04%	24.04%
0.70	24.00%	24.04%	24.04%
0.80	24.00%	24.04%	24.04%
0.90	24.00%	24.04%	24.04%
1.00	23.99%	24.04%	24.04%
Average	**25.88%**	**24.05%**	**24.05%**

Table B.8: Impact of the interference penalty on the WCET estimates for Matmult *Loop* (The WCET differences are relative to the WCET estimate of the LRU cache with interference penalty.)

Norm. Memory Size	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**
0.09	37.73%	-	-
0.18	29.09%	-	-
0.27	24.21%	24.75%	24.75%
0.36	24.00%	24.08%	24.08%
0.45	23.99%	24.05%	24.05%
0.55	23.99%	24.04%	24.04%
0.64	23.99%	24.04%	24.04%
0.73	23.99%	24.04%	24.04%
0.82	23.99%	24.04%	24.04%
0.91	23.99%	24.03%	24.03%
1.00	23.98%	24.02%	24.02%
Average	**25.72%**	**24.12%**	**24.12%**

Table B.9: Impact of the interference penalty on the WCET estimates for Puwmod (The WCET differences are relative to the WCET estimate of the LRU cache with interference penalty.)

Norm. Memory Size	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**
0.03	30.96%	-	-
0.05	30.96%	-	-
0.08	28.58%	-	-
0.10	28.04%	-	-
0.13	28.04%	-	-
0.15	28.04%	-	-
0.18	28.04%	-	-
0.20	28.04%	-	-
0.23	28.04%	-	-
0.25	28.04%	-	-
0.28	28.04%	-	-
0.30	28.04%	-	-
0.33	28.04%	-	-
0.35	28.04%	-	-
0.38	28.04%	-	-
0.40	28.04%	-	-
0.43	28.04%	-	-
0.45	28.04%	-	-
0.48	28.04%	-	-
0.50	28.04%	-	-
0.53	28.04%	-	-
0.55	28.04%	-	-
0.58	27.85%	-	-
0.60	27.85%	-	-
0.63	27.66%	-	-
0.65	27.66%	-	-
0.68	27.66%	-	-
0.70	27.66%	-	-
0.73	27.66%	-	-
0.75	27.66%	-	-
0.78	27.66%	-	-
0.80	27.66%	-	-
0.83	27.66%	-	-
0.85	27.66%	-	-
0.88	27.66%	-	-
0.90	27.66%	-	-
0.93	27.66%	-136.10%	-136.10%
0.95	27.66%	-136.10%	-136.10%
0.98	27.66%	-14.71%	35.87%
1.00	27.66%	35.87%	35.87%
Average	**28.04%**	**-62.76%**	**-50.12%**

Table B.10: Impact of the interference penalty on the WCET estimates for Puwmod *Split* (The WCET differences are relative to the WCET estimate of the LRU cache with interference penalty.)

Norm. Memory Size	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**
0.03	32.03%	-	-
0.07	32.03%	-	-
0.10	30.24%	-	-
0.13	29.85%	-	-
0.17	29.85%	-	-
0.20	29.85%	-	-
0.23	29.85%	-	-
0.27	29.85%	-	-
0.30	29.71%	-	-
0.33	29.71%	-	-
0.37	29.71%	-	-
0.40	29.71%	-	-
0.43	29.71%	-	-
0.47	29.71%	-	-
0.50	29.71%	-	-
0.53	29.71%	-	-
0.57	29.71%	-	-
0.60	29.64%	-	-
0.63	29.57%	-	-
0.67	29.57%	-	-
0.70	29.57%	-	-
0.73	29.57%	-	-
0.77	29.57%	-	-
0.80	29.57%	-	-
0.83	29.57%	-42.84%	-42.84%
0.87	29.57%	-27.23%	-27.23%
0.90	29.57%	9.19%	45.44%
0.93	29.57%	35.20%	45.44%
0.97	29.57%	35.20%	45.44%
1.00	29.57%	45.44%	45.44%
Average	**29.85%**	**9.16%**	**18.61%**

Table B.11: Impact of the interference penalty on the WCET estimates for Rspeed (The WCET differences are relative to the WCET estimate of the LRU cache with interference penalty.)

Norm. Memory Size	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**
0.04	30.58%	-	-
0.09	30.58%	-	-
0.13	30.10%	-	-
0.17	28.11%	-	-
0.22	28.11%	-	-
0.26	28.11%	-	-
0.30	28.11%	-	-
0.35	28.11%	-	-
0.39	28.11%	-	-
0.43	28.11%	-	-
0.48	28.11%	-	-
0.52	28.11%	-	-
0.57	25.84%	-	-
0.61	25.84%	-	-
0.65	25.84%	-	-
0.70	25.84%	-	-
0.74	25.84%	-	-
0.78	25.84%	-	-
0.83	25.84%	-	-
0.87	25.84%	-63.69%	-63.69%
0.91	25.84%	-3.95%	31.90%
0.96	25.84%	-3.95%	31.90%
1.00	25.84%	31.90%	31.90%
Average	**27.32%**	**-9.92%**	**8.00%**

Table B.12: Impact of the interference penalty on the WCET estimates for Rspeed *Split* (The WCET differences are relative to the WCET estimate of the LRU cache with interference penalty.)

Norm. Memory Size	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**
0.07	30.41%	-	-
0.13	30.41%	-	-
0.20	30.05%	-	-
0.27	29.41%	-	-
0.33	29.01%	-	-
0.40	29.01%	-	-
0.47	28.59%	-	-
0.53	28.59%	-	-
0.60	28.59%	-	-
0.67	28.59%	22.94%	22.94%
0.73	28.59%	22.94%	22.94%
0.80	28.59%	36.46%	49.99%
0.87	28.59%	43.47%	49.99%
0.93	28.59%	43.47%	49.99%
1.00	28.59%	43.47%	49.99%
Average	**29.04%**	**35.46%**	**40.97%**

Table B.13: Impact of the interference penalty on the WCET estimates for Sha (The WCET differences are relative to the WCET estimate of the LRU cache with interference penalty.)

Norm. Memory Size	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**
0.03	30.74%	-	-
0.05	26.40%	-	-
0.08	25.85%	-	-
0.10	25.85%	-	-
0.13	25.23%	-	-
0.15	25.17%	-	-
0.18	25.17%	-	-
0.20	25.17%	-	-
0.23	25.17%	-	-
0.25	24.55%	-	-
0.28	24.55%	-	-
0.30	24.07%	17.40%	17.40%
0.33	20.11%	3.16%	3.16%
0.35	20.10%	3.12%	3.12%
0.38	20.10%	3.12%	3.12%
0.40	20.08%	3.08%	3.08%
0.43	20.08%	3.08%	3.08%
0.45	20.08%	3.08%	3.08%
0.48	20.08%	3.08%	12.54%
0.50	20.08%	3.00%	30.71%
0.53	20.07%	-4.67%	30.69%
0.55	20.07%	2.17%	30.69%
0.58	20.07%	30.69%	30.69%
0.60	20.07%	30.69%	30.80%
0.63	20.07%	30.69%	30.86%
0.65	20.07%	30.69%	30.86%
0.68	20.07%	30.63%	30.86%
0.70	20.07%	30.63%	30.86%
0.73	20.04%	30.54%	30.78%
0.75	20.04%	30.54%	30.78%
0.78	20.04%	30.60%	30.78%
0.80	20.04%	30.71%	30.78%
0.83	20.04%	30.99%	30.78%
0.85	20.04%	30.99%	30.78%
0.88	20.04%	30.99%	30.78%
0.90	20.04%	30.99%	30.78%
0.93	20.04%	30.99%	30.78%
0.95	20.04%	30.99%	30.78%
0.98	20.04%	30.99%	30.78%
1.00	20.04%	30.99%	30.78%
Average	**21.74%**	**20.48%**	**23.96%**

Table B.14: Impact of the interference penalty on the WCET estimates for Ttsprk (The WCET differences are relative to the WCET estimate of the LRU cache with interference penalty.)

Norm. Memory Size	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**	**Norm. Memory Size**	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**
0.01	32.38%	-	-	0.51	21.65%	-	-
0.02	24.52%	-	-	0.52	21.65%	-	-
0.03	23.58%	-	-	0.53	21.65%	-	-
0.04	23.54%	-	-	0.54	21.65%	-	-
0.05	23.51%	-	-	0.55	21.65%	-	-
0.06	23.51%	-	-	0.56	21.65%	-	-
0.07	23.38%	-	-	0.57	21.65%	-	-
0.08	23.32%	-	-	0.58	21.65%	-	-
0.09	23.15%	-	-	0.59	21.65%	-	-
0.10	23.14%	-	-	0.60	21.65%	-	-
0.11	23.14%	-	-	0.61	21.65%	-	-
0.12	23.08%	-	-	0.62	21.65%	-	-
0.13	22.83%	-	-	0.63	21.64%	-	-
0.14	22.81%	-	-	0.64	21.64%	-	-
0.15	22.81%	-	-	0.65	21.63%	-	-
0.16	22.81%	-	-	0.66	21.57%	-	-
0.17	22.81%	-	-	0.67	21.52%	-	-
0.18	22.81%	-	-	0.68	21.52%	-	-
0.19	22.06%	-	-	0.69	21.52%	-	-
0.20	21.71%	-	-	0.70	21.52%	-	-
0.21	21.67%	-	-	0.71	21.52%	-	-
0.22	21.67%	-	-	0.72	21.52%	-	-
0.23	21.67%	-	-	0.73	21.52%	-	-
0.24	21.67%	-	-	0.74	21.52%	-	-
0.25	21.67%	-	-	0.75	21.52%	-	-
0.26	21.67%	-	-	0.76	21.52%	-	-
0.27	21.67%	-	-	0.77	21.52%	-	-
0.28	21.67%	-	-	0.78	21.52%	-	-
0.29	21.65%	-	-	0.79	21.52%	-	-
0.30	21.65%	-	-	0.80	21.52%	-	-
0.31	21.65%	-	-	0.81	21.52%	-	-
0.32	21.65%	-	-	0.82	21.52%	-128.19%	-128.19%
0.33	21.65%	-	-	0.83	21.52%	-55.34%	-55.34%
0.34	21.65%	-	-	0.84	21.52%	-55.34%	-55.34%
0.35	21.65%	-	-	0.85	21.52%	-55.34%	-55.34%
0.36	21.65%	-	-	0.86	21.52%	-9.63%	-9.63%
0.37	21.65%	-	-	0.87	21.52%	-9.63%	-9.63%
0.38	21.65%	-	-	0.88	21.52%	-9.63%	-9.63%
0.39	21.65%	-	-	0.89	21.52%	-9.63%	-9.63%
0.40	21.65%	-	-	0.90	21.52%	-9.63%	-9.63%
0.41	21.65%	-	-	0.91	21.52%	-9.63%	20.51%
0.42	21.65%	-	-	0.92	21.52%	-9.63%	20.51%
0.43	21.65%	-	-	0.93	21.52%	-9.63%	20.51%
0.44	21.65%	-	-	0.94	21.52%	-9.63%	20.51%
0.45	21.65%	-	-	0.95	21.52%	-27.22%	20.51%
0.46	21.65%	-	-	0.96	21.52%	7.08%	20.51%
0.47	21.65%	-	-	0.97	21.52%	7.08%	20.51%
0.48	21.65%	-	-	0.98	21.52%	7.08%	20.51%
0.49	21.65%	-	-	0.99	21.52%	4.57%	20.51%
0.50	21.65%	-	-	1.00	21.52%	22.26%	20.51%
				Average	**21.96%**	**-19.23%**	**-7.54%**

Table B.15: Impact of the interference penalty on the WCET estimates for Ttsprk *Loop* (The WCET differences are relative to the WCET estimate of LRU cache with interference penalty.)

Norm. Memory Size	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**	**Norm. Memory Size**	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**
0.01	32.37%	-	-	0.51	21.60%	-	-
0.02	24.52%	-	-	0.52	21.60%	-	-
0.03	23.57%	-	-	0.53	21.60%	-	-
0.04	23.53%	-	-	0.54	21.60%	-	-
0.05	23.50%	-	-	0.55	21.60%	-	-
0.06	23.50%	-	-	0.56	21.60%	-	-
0.07	23.37%	-	-	0.57	21.60%	-	-
0.08	23.32%	-	-	0.58	21.60%	-	-
0.09	23.14%	-	-	0.59	21.60%	-	-
0.10	23.13%	-	-	0.60	21.60%	-	-
0.11	23.13%	-	-	0.61	21.60%	-	-
0.12	22.89%	-	-	0.62	21.59%	-	-
0.13	22.81%	-	-	0.63	21.58%	-	-
0.14	22.80%	-	-	0.64	21.53%	-	-
0.15	22.80%	-	-	0.65	21.47%	-	-
0.16	22.80%	-	-	0.66	21.47%	-	-
0.17	22.80%	-	-	0.67	21.47%	-	-
0.18	22.33%	-	-	0.68	21.42%	-	-
0.19	22.02%	-	-	0.69	21.42%	-	-
0.20	21.67%	-	-	0.70	21.42%	-	-
0.21	21.63%	-	-	0.71	21.42%	-	-
0.22	21.63%	-	-	0.72	21.42%	-	-
0.23	21.63%	-	-	0.73	21.42%	-	-
0.24	21.63%	-	-	0.74	21.42%	-	-
0.25	21.63%	-	-	0.75	21.42%	-	-
0.26	21.63%	-	-	0.76	21.42%	-	-
0.27	21.63%	-	-	0.77	21.42%	-	-
0.28	21.60%	-	-	0.78	21.42%	-	-
0.29	21.60%	-	-	0.79	21.42%	-	-
0.30	21.60%	-	-	0.80	21.42%	-128,78%	-128,78%
0.31	21.60%	-	-	0.81	21.42%	-55.77%	-55.77%
0.32	21.60%	-	-	0.82	21.42%	-55.77%	-53.58%
0.33	21.60%	-	-	0.83	21.42%	-53.58%	-53.58%
0.34	21.60%	-	-	0.84	21.42%	-54.20%	-46.02%
0.35	21.60%	-	-	0.85	21.42%	-12.17%	-7.77%
0.36	21.60%	-	-	0.86	21.42%	-12.17%	-7.77%
0.37	21.60%	-	-	0.87	21.42%	-7.77%	-7.77%
0.38	21.60%	-	-	0.88	21.42%	-7.77%	-7.77%
0.39	21.60%	-	-	0.89	21.42%	-7.77%	-7.77%
0.40	21.60%	-	-	0.90	21.42%	-7.77%	22.44%
0.41	21.60%	-	-	0.91	21.42%	-7.77%	22.44%
0.42	21.60%	-	-	0.92	21.42%	-7.77%	22.44%
0.43	21.60%	-	-	0.93	21.42%	-36.41%	22.44%
0.44	21.60%	-	-	0.94	21.42%	-4.23%	22.44%
0.45	21.60%	-	-	0.95	21.42%	-4.23%	22.44%
0.46	21.60%	-	-	0.96	21.42%	8.98%	22.44%
0.47	21.60%	-	-	0.97	21.42%	8.98%	22.44%
0.48	21.60%	-	-	0.98	21.42%	8.67%	22.44%
0.49	21.60%	-	-	0.99	21.42%	22.00%	22.44%
0.50	21.60%	-	-	1.00	21.42%	24.24%	22.44%
				Average	**21.90%**	**-16.18%**	**-3.75%**

Table B.16: Impact of the interference penalty on the WCET estimates for Ttsprk *Split* (The WCET differences are relative to the WCET estimate of the LRU cache with interference penalty.)

Norm. Memory Size	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**	**Norm. Memory Size**	**ΔWCET I$ SHMC vs. I$ SEMC**	**ΔWCET I$ SHMC vs. D–ISP FIFO**	**ΔWCET I$ SHMC vs. D–ISP STACK**
0.01	32.44%	-	-	0.51	22.04%	-	-
0.03	24.80%	-	-	0.53	22.04%	-	-
0.04	24.19%	-	-	0.54	22.04%	-	-
0.05	23.92%	-	-	0.55	22.04%	-	-
0.06	23.90%	-	-	0.56	22.04%	-	-
0.08	23.87%	-	-	0.58	22.04%	-	-
0.09	23.87%	-	-	0.59	22.04%	-	-
0.10	23.87%	-	-	0.60	22.04%	-	-
0.11	23.74%	-	-	0.61	22.04%	-	-
0.13	23.74%	-	-	0.63	22.04%	-	-
0.14	23.58%	-	-	0.64	22.04%	-	-
0.15	23.51%	-	-	0.65	22.04%	-	-
0.16	23.34%	-	-	0.66	22.04%	-	-
0.18	23.29%	-	-	0.68	22.04%	-48.15%	-48.15%
0.19	23.29%	-	-	0.69	22.04%	-15.65%	-15.65%
0.20	23.18%	-	-	0.70	22.04%	-15.65%	-13.26%
0.21	23.18%	-	-	0.71	22.04%	-13.26%	-13.26%
0.23	23.18%	-	-	0.73	22.04%	-13.26%	-13.26%
0.24	23.18%	-	-	0.74	22.04%	-13.26%	-9.65%
0.25	23.18%	-	-	0.75	22.04%	4.08%	8.88%
0.26	23.18%	-	-	0.76	22.04%	4.08%	8.88%
0.28	23.18%	-	-	0.78	22.04%	8.88%	8.88%
0.29	23.18%	-	-	0.79	22.04%	8.88%	8.88%
0.30	22.79%	-	-	0.80	22.04%	8.88%	8.88%
0.31	22.36%	-	-	0.81	22.04%	8.88%	8.88%
0.33	22.15%	-	-	0.83	22.04%	8.88%	23.33%
0.34	22.12%	-	-	0.84	22.04%	8.88%	23.33%
0.35	22.11%	-	-	0.85	22.04%	8.88%	23.33%
0.36	22.10%	-	-	0.86	22.04%	8.88%	23.33%
0.38	22.10%	-	-	0.88	22.04%	8.88%	23.33%
0.39	22.10%	-	-	0.89	22.04%	0.45%	23.33%
0.40	22.08%	-	-	0.90	22.04%	12.88%	23.33%
0.41	22.06%	-	-	0.91	22.04%	12.88%	23.33%
0.43	22.04%	-	-	0.93	22.04%	17.70%	23.33%
0.44	22.04%	-	-	0.94	22.04%	17.70%	23.33%
0.45	22.04%	-	-	0.95	22.04%	17.70%	23.33%
0.46	22.04%	-	-	0.96	22.04%	18.90%	23.33%
0.48	22.04%	-	-	0.98	22.04%	22.53%	23.33%
0.49	22.04%	-	-	0.99	22.04%	22.53%	23.33%
0.50	22.04%	-	-	1.00	22.04%	25.02%	23.33%
				Average	**22.61%**	**5.08%**	**10.74%**

Curriculum Vitae

Personal Information

Name:	Stefan Metzlaff
Date of birth:	30th December 1980
Place of birth:	Rostock, Germany

Education

1999	Abitur
2000 – 2003	Study of Computer Science at University of Cooperative Education Stuttgart
2003	Diploma (BA) in Computer Science successfully completed
2003 – 2006	Study of Computer Systems in Engineering at University of Magdeburg
2006	Diploma in Computer Systems in Engineering successfully completed
since 2007	Ph.D. student at Department of Computer Science at University of Augsburg

Publications

Author

- S. Metzlaff and T. Ungerer (July 2012b). 'Replacement Policies for a Function-based Instruction Memory: A Quantification of the Impact on Hardware Complexity and WCET Estimates'. In: *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 112–121. DOI: 10.1109/ECRTS.2012.22

- S. Metzlaff and T. Ungerer (June 2012a). 'Impact of Instruction Cache and Different Instruction Scratchpads on the WCET Estimate'. In: *Proceedings of the 9th IEEE International Conference on Embedded Software and Systems (ICESS-2012)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 1442–1449. DOI: 10.1109/HPCC.2012.211

- S. Metzlaff, J. Mische and T. Ungerer (Nov. 2011b). 'A Real-Time Capable Many-Core Model'. In: *Proceedings of 32nd IEEE Real-Time Systems Symposium: Work-in-Progress Session*, pp. 21–24

- S. Metzlaff, I. Guliashvili, S. Uhrig and T. Ungerer (Feb. 2011a). 'A Dynamic Instruction Scratchpad Memory for Embedded Processors Managed by Hardware'. In: *Proceedings of the 24th International Conference on Architecture of Computing Systems (ARCS 2011)*. Vol. 6566. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 122–134. DOI: 10.1007/978-3-642-19137-4_11

- S. Metzlaff, S. Uhrig, J. Mische and T. Ungerer (Oct. 2008). 'Predictable Dynamic Instruction Scratchpad for Simultaneous Multithreaded Processors'. In: *Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture (MEDEA '08)*. New York, NY, USA: ACM, pp. 38–45. DOI: 10.1145/1509084.1509090

Associated Author

- M. Paolieri, J. Mische, S. Metzlaff, M. Gerdes, E. Quiñones, S. Uhrig, T. Ungerer and F. J. Cazorla. (2012). 'A Hard Real-Time Capable Multi-Core SMT Processor'. In: *ACM Transactions on Embedded Computing Systems (TECS)*. To be published.

- J. Wolf, M. Gerdes, F. Kluge, S. Uhrig, J. Mische, S. Metzlaff, C. Rochange, H. Cassé, P. Sainrat and T. Ungerer (Nov. 2011). 'RTOS Support for Execution of Parallelized Hard Real-Time Tasks on the MERASA Multi-core Processor'. In: *Computer Systems Science & Engineering* 26.6

- T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quiñones, M. Gerdes, M. Paolieri, J. Wolf, H. Cassé, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff and J. Mische (Oct. 2010). 'Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability'. In: *IEEE Micro* 30 (5), pp. 66–75. DOI: 10.1109/MM.2010.78

- J. Wolf, M. Gerdes, F. Kluge, S. Uhrig, J. Mische, S. Metzlaff, C. Rochange, H. Cassé, P. Sainrat and T. Ungerer (May 2010). 'RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-core Processor'. In: *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2010)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 193–201. DOI: 10.1109/ISORC.2010.31

- F. Kluge, J. Mische, S. Metzlaff, S. Uhrig and T. Ungerer (July 2007). 'Integration of Hard Real-Time and Organic Computing'. In: *ACACES 2007 Poster Abstracts*. Ghent, Belgium: Academia Press